Software-Qualität

Arbeiten des Bakkalaureatsseminars aus Angewandter Informatik

Sommersemester 2004

Seminarleitung:
Roland T. Mittermeir

Institut für Informatik-Systeme
Universität Klagenfurt
Juli 2004

TR ISYS-04/07-01

Bakkalaureatsarbeiten aus Informatik
Seminar aus Angewandter Informatik
Sommersemester 2004

Vorwort

Dieser Sammelband enthält die im Rahmen des Seminars aus Angewandter Informatik erstellten Arbeiten. Darunter sind acht Bakkalaureatsarbeiten und zehn von elf „klassischen" Seminararbeiten[1]. Die unterschiedliche Länge der einzelnen Beiträge entspricht den unterschiedlichen Vorgaben für Arbeiten unterschiedlicher Kategorien. Für Bakkalaureatsarbeiten wurden 6 ECTS Punkte vergeben. Seminararbeiten, die entsprechend der Studienordnung 2002 verfasst wurden, werden mit 4 ECTS Punkten gewichtet. Details der Abwicklung der Lehrveranstaltungen sowie Reflexionen über das erste Bakkalaureats-Seminar in Angewandter Informatik sind in einem abschließenden Beitrag des Lehrveranstaltungsleiters dargelegt.

Sämtliche Arbeiten wurden zum Globalthema *Software Qualität* verfasst. Die Reihenfolge der Präsentation richtet sich nach Art der Arbeit. Der Band beginnt mit den Bakkalaureatsarbeiten, da diese Raum hatten, ein selbst gewähltes Thema im Umfang von 10 Proceedings-Seiten zu bearbeiten. Es folgen die reinen Seminararbeiten. Hier hatten Zweierteams einen Raum von sechs Seiten und Einzelautoren vier Seiten zur Verfügung.

Selbstverständlich kann dieser Sammelband von Bakklaureats- bzw. Seminararbeiten keinen wie immer gearteten Anspruch auf Vollständigkeit stellen. Die Themen reflektieren Interessensgebiete der im abgelaufenen Semester am Seminar aus Angewandter Informatik teilnehmenden Studierenden. Dennoch wurde versucht, in der Reihenfolge innerhalb der beiden Gruppen wenigstens den Ansatz eines brüchigen roten Fadens zu finden.

Somit beginnt der Bakkalaureatsteil mit dem finalen Akt der Software-Entwicklung, einer Arbeit von ***Ingrid UNTERGUGGENBERGER*** über *Akzeptanztests.* Damit dieser positiv verlaufen kann, empfiehlt es sich, innerhalb des Entwicklungsprozesses nicht bloß auf dynamische sondern auch auf statische Qualitätssicherung zu achten. Optionen, die dafür bereits in frühen Entwicklungsphasen bestehen, bespricht ***Ursula DITTRICH*** in ihrer Arbeit *Die formale Inspektion – Eine spezielle Review-Technik*. Reviews durchzuführen setzt freilich voraus, dass innerhalb des Entwicklungsprozesses begutachtbare Dokumente entstehen. Dies versuchen die in jüngster Zeit propagierten agilen Entwicklungsprozesse zu vermeiden. Wie diese die daraus resultierenden Defizite ausgleichen, zeigt die Arbeit von ***Mario GRASCHL****, Qualitätssicherung bei agiler Software-Entwicklung*. Interessant mag zwischen den Arbeiten Graschls und Unterguggenbergers das Bild des in die Qualitätssicherung involvierten Benutzers bzw. Kunden sein. Advokaten dieser wie jener Methode sind aufgerufen, sich zu überlegen, welche inhaltlich-technischen Voraussetzungen kundenseitig gegeben sind, bevor man sich für ein konkretes Prozessmodell entscheidet.

Brigitte GAUSTER geht in institutioneller Sicht einen Schritt in Richtung Kunde. In der Arbeit *Software im Automobil – Qualitätssicherung durch MISRA* zeigt sie anhand des vielfältigen Einsatzes von Software in modernen Kraftfahrzeugen, dass eine gesamte Branche, die Automobilindustrie, branchenspezifische Standards entwickelte, um die Ergebnisse von

[1]) Eine Arbeit über *Remote Usability Testing von Web-Applikationen* erreichte leider nicht das für Aufnahme in diesen Sammelband erforderliche Niveau.

Forschungen zur Qualitätssicherung von life-critical Software auf ihre speziellen Bedürfnisse zu adaptieren und so zu verbreiten.

Eine völlig andere Perspektive auf Qualität nimmt eine zweite Gruppe von Arbeiten ein. Ausgehend von der Überlegung, dass ein Kunde letztlich nur dann ein Produkt der gewünschten Qualität geliefert bekommt, wenn auch die zum Angebot führende Kalkulation korrekt war, widmet sie sich der Aufwandsschätzung in frühen Projektphasen sowie der begleitenden Aufwandskontrolle und verbesserten Schätzung im Lauf des Projekts. *Daniela **ESBERGER*** beginnt diese Gruppe mit der Arbeit *Aufwandsschätzung am Beispiel COCOMO II*. Diese Arbeit wird durch den Aufsatz von ***Edmund URBANI**, Metriken zur statischen Analyse objektorientierten Source-Codes* kontrapunktisch ergänzt. Die von ihm angesprochenen Metriken können einerseits unmittelbar als Qualitätsindikatoren eingesetzt werden, andererseits stellen sie Daten zur Schätzung des zu erwartenden Testsaufwands zur Verfügung (siehe auch Arbeit von *Lassnig* und *Wernig-Pichler* im Seminarteil).

Freilich ist bei all diesen Schätzungen, insbesondere bei solchen, die sich auf große und lang laufende Projekte beziehen, zu beachten, dass Anforderungen nichts Statisches sind. Software-Entwicklung ist auch ein Prozess der Erkenntnisgewinnung. Anwender lernen im Laufe des Prozesses (mehr noch während des Einsatzes des neuen Systems) ihre Bedarfe besser zu strukturieren und damit auch besser zu artikulieren und Entwickler erkennen, das manches sich nicht so entwickeln lässt, wie ursprünglich vorgesehen. Daher ist es für die Qualität und Effizienz des Gesamtprozesses wesentlich, möglichst genau zu wissen, welche Anforderung sich an welcher Stelle des Entwurfs bzw. der Implementierung wieder findet, aber auch welche Globalanforderung ein bestimmtes Anforderungsdetail begründet. ***Kerstin JÖRGL*** geht in der Arbeit *Management of Requirements Traceability Problems* diesen Fragen nach, spannt das Problemfeld auf und referiert über Kosten-/Nutzen-Überlegungen, die gerade in diesem Bereich zentral sind, um zu vermeiden, dass man ein Fass ohne Boden öffnet. Der Kreis der Bakkalaureatsarbeiten schließt sich mit der Arbeit von ***Werner SÜHS*** über *Spannungen zwischen Benutzergewohnheiten und neuer Bedienbarkeit.* Sie verbindet Evolutionsaspekte von Software-Systemen mit Akzeptanzfragen. Sühs geht es allerdings nicht um jene Aspekte der Akzeptanz, die klassischer Weise in einem vorab fixierten formalen Akzeptanztest zu klären sind. Ihm geht es um die Probleme, die Benutzer mit der Einführung eines Neusystems im Wechselspiel zwischen Verlust des Gewohnten und Gewinn des Neuen zu bewältigen haben, Probleme, die von qualitätsbewussten Entwicklern zu antizipieren und weitestgehend zu minimieren sind.

Die Gruppe der reinen Seminararbeiten beginnt mit Ausführungen über den Software-Entwicklungsprozess. ***Gudrun EGGER***, eine Lehramts-Studierende, gibt in *CMM – Eine Einführung* einen Überblick über das Capability Maturity Model von SEI und zeigt dessen Entwicklung zu CMMI, während ***Birgit ANTONITSCH* und *Hubert GRESSL*** in *Entwicklung von ISO 9000* die Neustrukturierung der in gewisser weise dazu komplementären ISO 9000 Normenserie beschreiben und auf allgemeine Zertifizierungsüberlegungen eingehen. Mit Prozessaspekten die in interkontinental verteilt durchgeführen Projekten zu beachten sind, beschäftigt sich ***Marion KURY*** in *Aspekte des Software Engineering in globalen Projekten.*

An diese drei Arbeiten schließt ein Block mit Arbeiten zu Metriken an. Er wird durch Ausführungen zur *„Goal Question Metric“ Methode* von ***Michael JAKAB* und *Michael OFNER*** eröffnet. ***Johannes WERNIG-PICHLER* und *Mario LASSNIG*** präsentieren ihre Sicht auf *Objektorientierte Software-Metriken* und ***Daniel PEINTNER*** zeigt, in *Der optimale Software Release Zeitpunkt*, dass dieser anhand entsprechender Messungen und Prozessüberwachung tatsächlich ermittelt werden kann. Schließlich wenden sich ***Katharina FRITZ* und *Marina***

GLATZ in ihrer Arbeit *Zeitmanagement im individuellen Software-Entwicklungsprozess - unter spezieller Berücksichtigung schulischer Aspekte* der Erkenntnisgewinnung über persönliche Leistungsfähigkeiten im Bereich der Software-Entwicklung zu. Hierbei ist zu erwähnen, dass die beiden Lehramtsstudierenden mit Übertragung der allgemeinen Software-Engineering-Thematik auf den Schulbereich und die entsprechende kritische Auseinandersetzung mit dem Thema aus schulischer Sicht absolutes Neuland betreten haben.

Den abschließenden Block bilden drei Arbeiten zur Test-Thematik. ***Thomas FRANK*** zeigt in *Testen komponenten-basierter Software*, dass die zunehmend stärker eingesetzte Komponentenbauweise von Software auch auf den Testprozess Rückwirkungen hat. ***Stefan PERAUER*** **und** ***Robert SORSCHAG*** präsentieren mit *Objekt-Relations-Diagrammen* ein Hilfsmittel für *Effektives Testen objektorientierter Software* und ***Daniela INNERWINKLER*** **und** ***Gunar MÄTZLER*** zeigen in *Mutationentest – Objektorientierte Mutanten für Java-Programme*, wie die Wahl von Programmierparadigma und spezifischer Sprache in die Wahl der Mutanten zur Prüfung der Effektivität eines Satzes von Testdaten eingeht.

Damit findet der inhaltliche Teil dieses Sammelbandes seinen Abschluss. Da es sich hierbei um das erste Bakkalaureatsseminar in Informatik an der Universität Klagenfurt handelte und die Veranstaltung damit einen gewissen Erprobungscharakter hatte – der freilich durch die Mischung von Bakkalareatsstudierenden mit klassischen Seminaristinnen und Seminaristen nicht unerheblich beeinflusst wurde – wird die Gesamtdarstellung der studentischen Arbeiten durch eine methodische Reflexion des Lehrveranstaltungsleiters ergänzt. Möge diese in Zusammenschau mit der Qualität der studentischen Arbeiten künftigen Seminarleitungen als Anregung für allfällige Verbesserungen in der Gestaltung von Bakkalaureatsseminaren dienen.

Klagenfurt, 4. Juli 2004

Roland Mittermeir
Lehrveranstaltungsleiter

Seminar aus Angewandter Informatik

Globalthema: Software Qualität

Testen

Schlusswort



Bakkalaureatsarbeiten

Akzeptanztests

Unterguggenberger Ingrid
M# 9960490
iuntergu@edu.uni-klu.ac.at

Abstract

Die Bedeutung des Akzeptanztests liegt im rechtlichen Sinn darin, dass nach dem Test die Verantwortung für eine entwickelte Software an den Kunden oder Auftraggeber übergeht, da mit Bestehen des Abnahmetests die Entwicklungsphase abgeschlossen ist und die Wartungsphase beginnt. Deshalb ist es wichtig, dass der Abnahmetest korrekt durchgeführt wird. Dies ist allerdings nur dann möglich, wenn die entsprechenden Abnahmekriterien gut gewählt sind.

Genau diese Problematik wird in dieser Arbeit anhand von Richtlinien laut ISO 9000-3 bezüglich Benutzerakzeptanz und einiger Webseiten, die Dienstleistungen zu diesem Thema anbieten oder sonstiges Wissenswertes dazu zu bieten haben, erörtert. Außerdem werde ich versuchen, den Zusammenhang zwischen einem gut durchgeführten Abnahmetest und der Qualität des gesamten Softwareproduktes näher zu beleuchten.

Der Leser soll mit Hilfe dieser Arbeit die Wichtigkeit des Abnahmetests erkennen, und Grundlagen für die Ausführung und Einschätzung eines solchen erlangen.

1. Was ist ein Akzeptanztest?

Ein Akzeptanztest ist "Ein Test, ausgeführt für ein Software- oder Hardwaresystem, um dessen funktionale Leistungsfähigkeit bezüglich vordefinierter/festgelegter Erfordernisse zu evaluieren. Testet und bewertet Leistung, Funktion, Effizienz, Performance und Konformität gegenüber den bestimmten Aufgaben eines neu angeschafften Systems." [1]

In anderen Worten ausgedrückt: Ein Abnahmetest ist ein Test, der am Ende der Entwicklungsphase eines Systems, meistens vom Benutzer selbst, durchgeführt wird, und prüfen soll, ob das entwickelte Programm den im Vorhinein definierten Anforderungen entspricht. Fällt ein System beim Abnahmetest durch, geht es wieder zurück und wird überarbeitet. Besteht das System den Test, dann ist die Entwicklung abgeschlossen, die Verantwortung geht vom Verkäufer auf den Käufer über, und das System befindet sich in der Wartungsphase. [2]

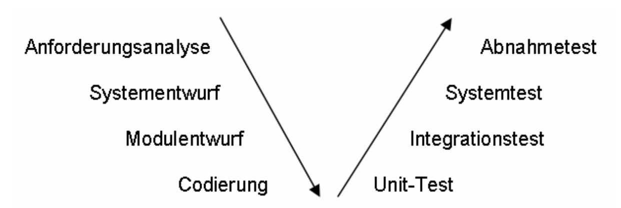


Abbildung 1: klassisches Testplanungsmodell [3]

In Abbildung 1 sieht man deutlich die Zusammenhänge zwischen den Entwicklungs- und Testschritten. Während der jeweils links stehenden Phase können die Testfälle geplant werden, die in der rechts stehenden Testphase dann abgeprüft werden. Hieraus erkennt man, dass die Testfälle für den Abnahmetest aus der Anforderungsanalyse hervorgehen [3]. Deswegen sollte man die Aussage von Dorothy Graham beherzigen: "Good requirements engineering produces better tests; good test analysis produces better requirements" [4].

Der Unit-Test sucht Fehler in den kleinsten testbaren Einheiten (=Units), wie etwa in Unterprogrammen, Funktionen, Methoden, … auf Basis der Implementierung. Der Integrationstest zeigt ob die Kombination von Komponenten inkorrekt oder inkonsistent ist, obwohl die einzelnen Komponenten für sich zufrieden stellend sind, dh er testet besondere Schnittstellenprobleme auf Basis des Modulentwurfs. Wohingegen der Systementwurf durch den Systemtest getestet wird. Genauer gesagt, untersucht der Systemtest Eigenschaften und Verhalten das nur durch Testen des ganzen Systems feststellbar ist wie zB Performance, Sicherheit oder Zuverlässigkeit.

Abnahmetests unterscheiden sich von den anderen Testarten darin, dass sie nicht versuchen Fehler zu finden, sondern im Gegenteil nachweisen wollen, dass alle Anforderungen erfüllt werden. Die Testfälle des Akzeptanztests sind meistens so aufgebaut, dass sie die

Erfüllung eines einzelnen Abnahmekriteriums belegen [3].

Ein Beispiel für einen Akzeptanztest wäre: "Das System kennt die Kundennummern 100, 200 und 300. Bei Eingabe einer dieser Nummern wird die Bestellung ins System übernommen. Bei Eingabe einer anderen Nummer, wird die Bildschirmseite für die Neuerfassung eines Kunden angezeigt."

Zum Abnahmetest gehört auch die Vollständigkeit (wurden alle Funktionen der Anforderungsdefinition realisiert?) und die Usability (sind Fehlerausgaben angemessen, wurden Ausgaben aufbereitet, Format, Stil, Abkürzungen, gibt es genügend Redundanz in der Eingabe, Optionen die nicht benutzt werden, ist die Benutzerschnittstelle einheitlich, …).

2. Wofür benötigt man Abnahmetests?

Wie schon erwähnt ist der Akzeptanztest ein Indikator dafür, ob ein Programm seinen Anforderungen entspricht, oder nicht. Solange der Akzeptanztest nicht bestanden wird, ist das System noch nicht fertig und der Entwickler nicht von seiner Verantwortung entbunden. [2]

Akzeptanztests verlieren ihren Wert nicht, auch wenn der Kunde das System als implementiert abgenommen hat. Sie sind weiterhin erforderlich, um sicherzustellen, dass weitere Änderungen am System bereits vorhandene Funktonalität nicht modifiziert haben, somit erhöht sich die Qualität nachfolgender Tests. Außerdem spart man dadurch Zeit und Geld. [5]

Es gibt die verschiedensten Einteilungen von Akzeptanztests. Laut Spillner [6] wird der Abnahmetest in den 'Test auf vertragliche Akzeptanz' (in [7] 'Produktions-Abnahmetest'), in den 'Test auf Benutzerakzeptanz' (in [7] 'funktionaler Abnahmetest') und in den Feldtest unterteilt, wobei er letzteren noch einmal in 'Alpha Test' und 'Beta Test' aufteilt. Norman Parrington [8] und Edward Kit [9] hingegen unterscheiden generell nur zwischen 'Alpha Test' und 'Beta Test'. Ich verwende hier die Einteilung von Spillner [6].

2.1 Test auf vertragliche Akzeptanz

Hierbei prüft der Kunde, ob die vorliegende Software den Vertrag bezüglich der dort vereinbarten Systemeigenschaften erfüllt. Es wird also die Software auf Mängel aus Kundensicht untersucht. Um hier eine definierte Entscheidungsbasis zu haben, müssen die Abnahmekriterien zuvor klar und präzise im Vertrag definiert worden sein. Bevor der Hersteller die fertige Software dem Kunden zur Abnahme vorlegt, sollte allerdings der Systemtest im eigenen Haus schon gezeigt haben, dass die Abnahmekriterien durch das System überhaupt erfüllt werden können. Dennoch kann es durchaus passieren, dass beim Hersteller erfolgreich durchgeführte Tests in der Kundenumgebung scheitern, was letztlich zeigt, dass der Systemtest den Abnahmetest nicht ersetzen darf.

2.2 Test auf Benutzerakzeptanz

Dieser Test ist nur dann sinnvoll, wenn der Anwender der Software nicht gleichzeitig auch der Kunde ist. Bei diesem Test wird das System durch seine späteren Anwender, und nicht nur durch den Auftraggeber, in Hinblick auf die Erfüllung ihrer Erwartungen (zB bezüglich Benutzerfreundlichkeit) getestet. Wird das System durch unterschiedliche Benutzergruppen verwendet, so sollte jede Gruppe durch einen eigenen Akzeptanztest repräsentiert sein, um so möglichst viele verschiedene Nutzungsszenarien abzudecken. Zu erwähnen ist auch, dass in Hinblick auf den Test auf Benutzerakzeptanz die Erstellung von Prototypen in den frühen Projektphasen, wie zB nach der Anforderungsanalyse, sinnvoll ist, um Problemen während des Akzeptanztests vorzubeugen und so gegen Projektende keine bösen Überraschungen zu erleben.

2.3 Feldtest

Soll die zu liefernde Software in sehr vielen verschiedenen Umgebungen betrieben werden, wird dem Systemtest ein Feldtest nachgeschalten, dessen Ziel es ist, Einflüsse aus nicht vollständig bekannten oder nicht spezifizierten Umgebungen zu erkennen und gegebenenfalls zu beheben.

2.3.1 Alpha Test. Dieser Test ist der erste durchgeführte Test. Er wird in den Räumen des Entwicklers von einem oder mehreren Anwendern in Gegenwart des Entwicklers durchgeführt.

2.3.2 Beta Test. Der Beta Test ist der Probebetrieb einer Vorabversion eines Softwareprodukts durch repräsentative Anwender in der Einsatzumgebung des Kunden. Dabei werden Probleme und Fehler protokolliert und an den Anbieter zurückgemeldet.

Dieses Vorgehen ist für den Hersteller besonders dann interessant, wenn die Verschiedenheit der möglichen Einsatzumgebungen mit denen die Software im realen Betrieb konfrontiert sein wird, nur schwer abgeschätzt werden kann, oder wenn aufgrund der Anzahl der möglichen Einsatzumgebungen ein Test mit den

verschiedenen Konfigurationen im Hause des Herstellers aus Kostengründen nicht möglich ist.

3. Was sagt ISO 9000-3 über Akzeptanztests?

Dieses Kapitel wurde zur Gänze dem Buch ISO 9000-3 [10] entnommen.

3.1 Allgemeines

Der Akzeptanztest wird vom Käufer durchgeführt und sollte gut geplant werden. Deshalb ist es ratsam einen Akzeptanztestplan aufzustellen, der den Zeitplan, die Ressourcen, Rollen, Verantwortlichkeiten, Testfälle und Erfolgskriterien im Vorhinein plant. Auch der Entwurf der Testfälle ist sehr wichtig und sollte genau vorgenommen werden. Käufer und Verkäufer müssen in der Gültigkeit der Akzeptanztestfälle übereinstimmen; genauso wie in der Genauigkeit der Prüfung der Ergebnisse dieser Testfälle. Die Auslieferung, Installation und Verantwortlichkeiten des Schlüsselpersonals sind im Akzeptanztest inbegriffen.

Der Käufer sollte die Tests selbst entwickeln um sicherzustellen, dass das zu liefernde Produkt den Anforderungen und Betriebszielen entspricht, und sich nicht auf die Testfälle und –ergebnisse des Verkäufers verlassen. Allerdings sollten diese Akzeptanztests vom Verkäufer reviewed werden, und beide, der Käufer und der Verkäufer müssen der Korrektheit der Tests zustimmen. Des Weiteren muss ein Prozess festgelegt werden, um Probleme, die während des Akzeptanztests auftauchen, zu lösen. Das umfasst die Problemidentifikation, -verfolgung und die Auslieferung von festgesetzten Software-Versionen, um Verzögerungen und Verwirrung an kritischen Punkten zu vermeiden.

Der Vertrag sollte die Verantwortlichkeiten zwischen Käufer und Verkäufer in Bezug auf den Akzeptanztest festlegen. Das Software Prozess Handbuch (dieses definiert den Software-Entwicklungsprozess) des Verkäufers sollte Abnahmetests als eine eigene Entwicklungsstufe beinhalten. Dieses sollte in den Allgemeinen Bedingungen die Aktivitäten, die während des Akzeptanztests auftreten, identifizieren, und sich nicht so sehr auf die Tests selbst konzentrieren (da diese im Verantwortungsbereich des Käufers liegen), sondern eher auf die Unterstützung, die der Verkäufer dem Käufer während dieser Phase eventuell erweisen muss.

Wenn der Käufer die Unterstützung des Verkäufers zur Durchführung des Akzeptanztests oder zur Lösung von Problemen in dieser Phase benötigt, sollte dies im Softwareentwicklungsplan durch einen Zeit- und Ressourcenplan reflektiert werden. Die Prioritäten für Probleme und akzeptable Antwortzeiten für die verschiedenen Kategorien sollten identifiziert und im Vertrag festgehalten werden.

Die Akzeptanzkriterien des Käufers sollte als 'Deliverables' des Käufers in den Vertrag aufgenommen werden. Der Verkäufer sowie auch der Käufer sollten sich beide bewusst sein, dass ein sich entwickelndes Set von Akzeptanzkriterien, egal ob basierend auf 'Evolving Requirements' oder auf einem wachsenden Verständnis des Käufers gegenüber dem Produkt, das Budget zum Explodieren bringen kann. Das Käufer- und Verkäufer-Management sollten diesen Teil des gesamten Plans und Prozesses, aufgrund der Auswirkung die er auf das Budget, den Zeitplan und das Käufer - Verkäufer Verhältnis haben kann, im Auge behalten.

3.2 Akzeptanztestplan

Der Akzeptanztestplan sollte am Beginn der Akzeptanztestphase aufgestellt werden, und folgende Punkte enthalten:

- Zeitplan
- Evaluierungsvorgang
- Soft-/Hardware Umgebung und Ressourcen
- Akzeptanzkriterium

Bei der Identifikation dieser Punkte sollte der Verkäufer dem Käufer assistieren. Der Grad dieser Hilfeleistung kann stark variieren. Tatsächlich kann (und wird oftmals auch) alles in dieser Sektion vom Käufer allein bestimmt werden, mit nur einer kleinen oder gar keinen Rolle des Verkäufers, aber genauso auch umgekehrt.

Für die Beschreibung der nächsten Punkte nehmen wir an, dass Käufer und Verkäufer sich die Verantwortung teilen. Der springende Punkt ist allerdings, dass der Verkäufer sicherstellen muss, dass der Akzeptanztestplan des Käufers wirklich das Produkt testet und dass Probleme, die während des Planes auftauchen, wirklich am Produkt liegen, und nicht an einem schlechten Akzeptanztest.

3.2.1. Zeitplan. Jede Aktivität benötigt einen Zeitplan, so dass Ressourcen verfügbar gemacht und verwandte Aktivitäten synchronisiert werden können. Je weiter weg vom vollständigen Prozess ein Plan kreiert wird (und Abnahmetests sind so weit weg wie man nur kommen kann), umso ungenauer wird der Plan sein. Deswegen müssen Käufer und Verkäufer zusammenarbeiten um den Zeitplan für den Akzeptanztest aufzu-

stellen und ihn entsprechend berichtigen, wenn die Realität den Entwicklungsplan stört. Der Startpunkt und die Dauer sind Schätzungen und sollten daher mit einer gewissen Flexibilität betrachtet werden.

3.2.2. Evaluierungsvorgang. Der Verkäufer muss mit dem Käufer zusammenarbeiten um sicherzustellen, dass der Käufer genug Verständnis für das Produkt hat um den Abnahmetest durchzuführen. Beide sollten darin übereinstimmen, dass die entworfenen Prozeduren auch ein legitimer Test des Produktes sind.

3.2.3. Software/Hardware Umgebung und Ressourcen. Der Akzeptanztestplan sollte die Hardware- und Softwareumwelt identifizieren, die benötigt wird, um das Produkt zu testen. Zusätzlich könnte der Käufer die Unterstützung des Verkäufers zur Konfiguration des fertigen Softwareproduktes oder der Hardware/Software Umgebung des Abnahmetests gebrauchen.

3.2.4. Akzeptanzkriterien. Der Käufer könnte Schwierigkeiten mit der Identifikation der Akzeptanzkriterien für das Produkt haben. Das kann einerseits am Erfahrungsmangel des Käufers liegen, oder aber auch an einem eingeschränkten Verständnis für das fertige Produkt. Der Verkäufer kann dem Käufer bei der Entwicklung der Kriterien helfen. Diese Hilfe minimiert Missverständnisse, die das Bestehen der einzelnen Abnahmekriterien betreffen. Der Käufer muss allerdings beachten, dass er trotzdem allein verantwortlich für die Genauigkeit der Kriterien ist. Dazu kommt, dass, wenn Zeitplan und Budgets einmal auf Grund dieser Kriterien aufgestellt wurden, alle Ausgaben des Verkäufers, die durch nachträgliche Änderungen an den Akzeptanzkriterien entstehen, vom Käufer getragen werden müssen.

Genau beim Punkt der nachträglichen Änderung von Anforderungen kommt noch ein zusätzlicher Aspekt dazu: Extreme Programming (kurz XP). XP lässt sich mit einem Puzzle vergleichen: Kunden und Entwickler sind Teil eines Teams, dessen Ziel es ist hochqualitative Software zu entwickeln, was sich ja nicht von anderen Methoden der Softwareentwicklung entscheidet. Allerdings werden bei XP nur kleine Teile entwickelt und dann zusammengefügt, und nach jeder Iteration ein Treffen mit dem Kunden angesetzt, um zu prüfen ob das System seinen Anforderungen entspricht bzw. ob der Kunde mittlerweile zusätzliche Aspekte erkannt hat. Der Vorteil von XP liegt darin, dass es für SW-Projekte eingesetzt werden kann, in denen noch nicht alle Anforderungen von vornherein klar spezifiziert werden können bzw. für Projekte, deren Anforderungen sich laufend ändern. [17]

4. Wie wird ein Akzeptanztest durchgeführt?

In diesem Abschnitt werde ich eine Methode zur Durchführung eines Akzeptanztests vorstellen, die William Perry in seinem Buch 'Effective Methods for Software Testing' [11] auf den Seiten 238 bis 244 ausführlich beschreibt.
Sein Prozess besteht aus fünf Schritten, die in Abbildung 2 veranschaulicht werden:

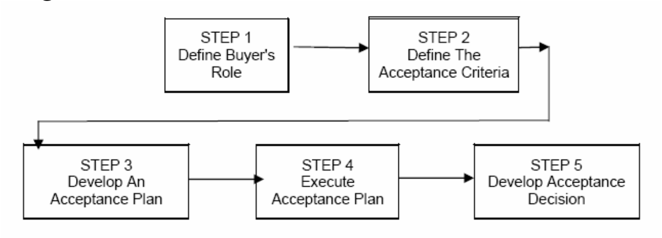


Abbildung 2: Akzeptanztestprozess [11]

4.1. STEP 1: Bestimme die Rolle des Käufers

Verantwortlich für die Software Akzeptanz ist der Käufer, in dessen Verantwortungsbereich auch die folgenden Punkte liegen:

- Sicherstellung, dass der/die Anwender in die Entwicklung der Anforderungen und der Akzeptanzkriterien einbezogen werden
- Identifizierung des fertigen Produktes, der Akzeptanzkriterien und des Zeitplans
- Planung, wie und von wem jede Aktivität während des Akzeptanztests ausgeführt wird
- Planung der Ressourcen, um Informationen für die Akzeptanzentscheidung zu sammeln
- Ausreichend Zeit für die späteren Anwender zur Verfügung stellen, damit sie das Produkt untersuchen und auswerten können, vorzugsweise vor dem Akzeptanztestreview
- Vorbereitung des Akzeptanztestplans
- Ausführung der endgültigen Akzeptanztestaktivitäten bei der Auslieferung, inklusive des formalen Akzeptanztests
- Treffen der Akzeptanzentscheidung für jedes Produkt.

Es besteht jedoch die Möglichkeit einige Punkte an einen Akzeptanztestmanager abzutreten. Dieser Manager kann ein Benutzer, Entwickler oder auch eine dritte Partei sein.

4.2. STEP 2: Identifiziere die Abnahmekriterien

Der Käufer muss die Kriterien, denen die Software entsprechen muss, festlegen. Idealerweise sind diese Kriterien in der Anforderungsspezifikation schon enthalten. Als Vorbereitung zur Entwicklung der Akzeptanzkriterien sollte der Käufer:

- alles über die Applikation, für die das System bestimmt ist, wissen
- die Risiken und Vorteile derjenigen Softwareentwicklungsmethode verstehen, die zur Bildung des Systems verwendet wird
- die Konsequenzen verstehen, die das Hinzufügen von neuen Funktionen zur Verbesserung eines existierenden Systems nach sich ziehen kann.

4.3. STEP 3: Entwickle den Akzeptanztestplan

Der erste Schritt zu effektiver Softwareakzeptanz ist die Entwicklung eines Akzeptanztestplans, allgemeiner Projektpläne und vertraglicher Anforderungen, um sicherzustellen, dass die Anforderungen der/s Anwender/s korrekt repräsentiert und auch ganz abgedeckt werden.

Der Akzeptanztestplan bringt eine Übersicht über die Tätigkeiten während des Akzeptanztests und soll dafür sorgen, dass die Ressourcen dafür auch im Projektplan enthalten sind. Allerdings kann der Plan während der Entwicklung noch ergänzt werden, und muss nicht von Anfang an schon alle Details enthalten.

Im Akzeptanztestplan enthalten sein sollten:

4.3.1. Projektbeschreibung. Typ des Systems, Lebenszyklusmethodik, Benutzergemeinde des gelieferten Systems, die Hauptaufgaben des fertigen Systems, externes Interface des Systems, erwarteter normaler Gebrauch, potentieller Missbrauch, Risiken, Bedingungen, Standards und Praktiken.

4.3.2. Benutzerverantwortlichkeiten. Organisation und Verantwortlichkeiten für die Akzeptanzaktivitäten, Ressourcen und Zeitplanung, Annehmlichkeitsanforderungen, Anforderungen für automatischen Support, spezielle Daten, Standards, Konventionen, Updates und Reviews des Akzeptanzplans und verwandter Produkte.

4.3.3. Administration. Anomalieberichte, Änderungskontrollen, Kommunikation zwischen Entwickler und Managementorganisationen.

4.3.4. Akzeptanzbeschreibung. Ziele für das gesamte Projekt, Zusammenfassung der Akzeptanzkriterien, hauptsächliche Akzeptanzaktivitäten und Reviews, Informationsbeschaffung, Typen von Akzeptanzbeschreibungen, Verantwortlichkeiten für die Akzeptanzbeschreibungen.

4.3.5. Akzeptanztestreview. Produkte für die Akzeptanz, Ziele für jeden Review, Akzeptanzkriterien, Bezugsquelle für Zusatzinformationen zum Produkt, Akzeptanzanforderungen, Test- und Untersuchungstechniken und benötigter automatischer Support.

4.3.6. Endgültiges Akzeptanztesten. Testplan und Akzeptanzkriterien, Testfälle und Prozeduren, Testergebnisse und Analysen, Werkzeugakquisition und Überprüfung, Belegschaft.

4.4. STEP 4: Führe den Akzeptanztestplan aus

Das Ziel dieses Schrittes ist zu bestimmen, ob die Akzeptanzkriterien von einem gelieferten Produkt getroffen wurden. Das kann durch Reviews erreicht werden, die immer wieder Zwischenprodukte betrachten. Es kann aber auch das Testen des ausführbaren Softwaresystems beinhalten. Die Entscheidung welche Technik verwendet wird, hängt davon ab, wie kritisch und wie groß die Software ist, genauso wie von den involvierten Ressourcen und der Zeitspanne, in der das Produkt entwickelt werden soll.

Der Review über die Ergebnisse des Akzeptanztests ist meistens der letzte Schritt im Software Akzeptanzprozess. Ein Vergleich von Schlüsselpunkten zwischen dem Akzeptanztestplan und den tatsächlich ausgeführten Aktivitäten zeigt, in welch detaillierten technischen Level der Käufer einbezogen wird. Diese Schlüsselpunkte inkludieren Planung und administrative Verantwortlichkeiten, Ziele, Annäherungen, örtliche und automatische Anforderungen, Mitarbeiterverantwortlichkeiten und Dokumentation für das Software Akzeptanztesten.

4.5. STEP 5: Triff die Akzeptanzentscheidung

Endgültige Softwareakzeptanz bedeutet normalerweise, dass der Vertrag und das Projekt komplettiert wurden, mit Ausnahme von Mängeln bei der Akzeptanz. Die endgültige Bezahlung der Software wird getätigt und der Entwickler hat keine weiteren Entwicklungsverpflichtungen.

Softwareakzeptanz ist ein vertraglicher Prozess. Bestimmte Arten von Software müssen die Akzeptanz

bestehen, noch bevor die Anforderungen vollständig spezifiziert wurden. Dazu gehören:

- Software, die die Entwicklung des Systems unterstützt
- Software, um das System zu betreiben
- Existierende Software für die Eingliederung in das System

Rekapitulieren wir noch einmal die wichtigsten Punkte:

1. Der/die Entwickler muss/müssen den Akzeptanzkriterien, die der Käufer entwickelt hat, zustimmen.
2. Der Käufer muss die Akzeptanzkriterien basierend auf den Systemanforderungen definieren.
3. Andere Projektcharakteristiken, wie spezielle Methodiken, müssen in die Definition der Akzeptanzkriterien einbezogen werden.
4. Akzeptanzentscheidungen basieren auf Analysen und Reviews des Produktes, sowie auf den Ergebnissen der Software Product Assurance Aktivitäten.
5. Der Käufer muss das Software-Akzeptanzprogramm sorgfältig planen und managen, um sicherzustellen, dass ausreichend Ressourcen für die Akzeptanztestaktivitäten zur Verfügung stehen.
6. Der Käufer muss detaillierte Planung für die Akzeptanztests in der ersten Planung des Software Akzeptanzprogramms einschließen.
7. Software Akzeptanz benötigt Ressourcen und Zustimmung des Käufers vom Start des Projekts an.
8. Als ein interaktiver Prozess, der speziell den Anwender involviert, resultiert die Vervollständigung des Abnahmetests in der gelieferten Software, die den Benutzern die Services bietet, die sie benötigen.

5. Welche Probleme können bei Akzeptanztests auftreten?

Wie in jedem anderen Bereich, können auch hier Schwierigkeiten auftreten.

Zum Beispiel können Unklarheiten auftreten, wenn der Kunde (=Auftraggeber der Software) nicht gleichzeitig auch der spätere Benutzer ist. Auf dieses Problem wird in Abschnitt 6.4 noch einmal eingegangen. Solche Unklarheiten können nur dadurch verhindert werden, dass nicht nur der Auftraggeber, sondern auch der spätere Anwender in die Anforderungsanalyse miteinbezogen wird. Somit wären wir wieder beim Thema XP (siehe Kapitel 3, letzter Absatz).

Schwierigkeiten ergeben sich auch, wenn die Abnahmekriterien zu ungenau ausgewählt wurden. Hierbei stellt sich die Frage, wer dafür die Verantwortung trägt: ist es der Käufer, der nicht genau formuliert hat was er will, oder ist es vielleicht der Verkäufer, der nicht genau nachgefragt hat? Oder haben hier beide eine Teilschuld? Ist der Verkäufer jetzt verpflichtet Anpassungen an die wirklichen Anforderungen ohne Aufpreis durchzuführen, oder sind diese Änderungen vom Käufer zu bezahlen? Um hier lange Rechtsstreitigkeiten bezüglich Schuld zu vermeiden, ist es notwendig die Verantwortlichkeiten im Vorhinein klarzustellen, worauf auch in [11] und [10] ausdrücklich hingewiesen wird.

Aber auch wenn die Abnahmekriterien genau gewählt wurden, kann unter Umständen ihre Prüfung problematisch werden. zB 'Die Suche nach einem Kunden in der Datenbank soll unabhängig vom Suchkriterium nicht länger als fünf Sekunden dauern.' Ein diesbezüglicher Abnahmetest müsste eigentlich alle Kombinationen aller möglichen Suchkriterien zum Zugriff auf die Kundendatenbank bilden und die jeweiligen Zugriffszeiten testen. Bei 10 Suchkriterien wären das bereits 1023 mögliche Kombinationen. Da man dies unter Einhaltung eines vertretbaren Zeitaufwandes nur durch ein Programm testen kann, werden bei solchen Kriterien entweder nur Stichprobentests durchgeführt, oder es wird eine längere Testperiode vereinbart, in der bereits mit dem Programm gearbeitet wird, und hofft dabei, dass sich Ausreißer in den kritischen Bedingungen innerhalb dieser Phase zeigen. [3]

Des Weiteren sollte verhindert werden, dass ein vertraglich vereinbarter Akzeptanztest zu einem Prüfstein für die Praxistauglichkeit der Software umfunktioniert wird. Denn dafür gibt es den System- und Installationstest. [12]

Ein anderer nicht zu vergessender Aspekt ist: 'Was nicht im Anforderungskatalog explizit formuliert ist, ist auch nicht Gegenstand eines vertraglichen Abnahmetests' [3]. Was uns wiederum auf den Punkt der genau zu formulierenden Akzeptanzbedingungen führt.

Auch sollte man nicht die Zielsetzungen des Abnahmetestens (Testen des operationalen Betriebs des Systems) außer Acht lassen [13], und außerdem sollte dafür gesorgt werden, dass sich die Endbenutzer bei Test wirklich an die geplanten Testfälle halten, denn durch eine nur willkürliche Benutzung des Systems können Fehler übersehen werden. [13]

Eine problematische Situation entsteht auch, wenn der Kunde zwei Wochen vor Durchführung der Akzeptanztests plötzlich seine Anforderungen ändert. Denn dadurch kann das Budget bezüglich Zeit bzw.

Geld stark in die Höhe schießen. Verhindern kann man solche Situationen nur, wenn der Kunde in die Anforderungsanalyse miteinbezogen wird. [4]

6. Akzeptanztests in der Praxis?

Es gibt sogar Unternehmen, zu deren Dienstleistungen unter anderem projektbegleitende Qualitätssicherung und Vorbereitungen zum Abnahmetest gehören. Solche Unternehmen sind die beiden deutschen Dienstleister, die TÜV Informationstechnik GmbH (kurz TÜVIT) [14] und die Software Quality Systems AG (kurz SQS) [15], die ich in diesem Kapitel kurz vorstellen werde. Zu den praktischen Beispielen gehört auch ein Internetforum, auf das ich bei meinen Internetrecherchen gestoßen bin: das WIKI WIKI Web [12]. Hier tauschen Menschen (vor allem Software Entwickler) aus aller Welt Informationen über ihre Erfahrungen aus. Und unter anderem gibt es hierbei auch eine Seite zum Thema Akzeptanztests, die hier behandelt wird. Den Abschluss meines Ausflugs in die Praxis macht die Homepage des Governments of British Columbia, oder besser gesagt des 'Ministry for Sustainable Resource Management' [5], das Richtlinien für Abnahmetests herausgegeben hat. Diese Richtlinien werde ich etwas genauer durchleuchten.

6.1. TÜV Informationstechnik GmbH

Die TÜV Informationstechnik GmbH (kurz TÜViT) ist ein Unternehmen der RWTÜV-Gruppe und ging aus dem 1990 gegründeten Institut für Informationstechnik hervor. TÜVIT wurde 1996 eine eigenständige GmbH, die die Durchführung von Reviews, die Testplanung, die Erstellung der Testspezifikation, die Testfallbeschreibung, die Testdurchführung sowie den Aufbau wiederholbarer, auf Wunsch auch automatisierter, Regressions- und Abnahmetests anbietet, und dazu erfahrene Testexperten oder ein komplettes Team inklusive Testmanager zur Verfügung stellt. Für TÜVIT besteht der Abnahmetest allerdings nur aus einer Zusammensetzung des Funktions-, Integrations-, System-, Installations- und Leistungstests.

Die Tests werden im TÜVIT-TestCenter (TTC) durchgeführt. Dort werden auch Mitarbeiter ausgebildet, Methoden weiterentwickelt und Testtools erprobt. Zu den weiteren Dienstleistungen des TTC gehören auch Workshops und Schulungen zu Testmethoden und Testwerkzeugen sowie die Bereitstellung kompletter Testteams, mit Erfüllung der Aufgaben des Testmanagements, Testplanung, Testspezifikation und Testdurchführung. Es gibt keine adhoc-Tests, da systematisch und strukturiert vorgegangen wird.

Der Vorteil für den Kunden liegt hier darin, dass die TÜVIT Erfahrungen und Know-How durch die Durchführung zahlreicher Testprojekte gesammelt hat.

Dass TÜVIT eine hohe praktische Bedeutung hat, sieht man meiner Meinung nach auch daran, dass andere Unternehmen, unter anderem Microsoft Deutschland (www.microsoft.de) sowie die bremen online services GmbH & Co KG (www.bos-bremen.de, ein Unternehmen das E-Goverment-Lösungen für die deutsche Regierung realisiert) Wert auf TÜV-Zertifizierung legen, und das auf ihren Homepages groß herausstreichen. Während sich die TÜVIT an den tatsächlichen Kunden richtet, wendet sich die nächste Firma eher an die Entwickler.

6.2. Software Quality Systems AG

Eine andere renommierte Firma auf dem Gebiet des Qualitätsmanagements ist die Kölner SQS Software Quality Systems AG. SQS bietet eine umfassende Auswahl an skalierbaren und projektbegleitenden Services, die auf die branchenspezifischen Bedürfnisse der Unternehmen zugeschnitten werden.

Diese Firma wirbt mit Slogans wie 'Überlassen Sie das Testen nicht Ihren Kunden', 'Kosten, Zeit und Risiko minimieren…' und ähnliches.

Die Software Quality Systems AG führt Tests (darunter auch Anwendungstests, Funktionstests, Systemtests, …) auf allen Ebenen des Software-Entwicklungsprozesses durch. Dabei wird die Testumgebung nach spezifischen Erfordernissen des Kunden (ist in diesem Fall der Entwickler) aufgebaut. Nach dem erfolgreichen Test im TestLab der SQS, wird das Teilsystem auf Wunsch ausgeliefert, in der Umgebung des Benutzers implementiert und die Mitarbeiter in der Durchführung der Wiederholungstests angeleitet. Damit können in Zukunft anstehende Wartungstests in einer praxiserprobten Testumgebung selbständig durchgeführt werden.

Nach eigenen Angaben ist SQS der führende Dienstleister für Software-Testen und Software-Qualitätsmanagement in Europa.

Basierend auf rund 20 Jahren Projekterfahrung setzt die SQS-Gruppe Standards für Methoden und Technologien des Software-Testens und Qualitätsmanagements. Ihre Leistungen decken Bereiche wie die Großrechnerwelt, Client/Server-Anwendungen und auch Internet-Lösungen ab.

SQS ist auch eines der größten Schulungsunternehmen für SW-Testen und SW–Qualitätsmanagement in Europa und hält immer wieder Seminare, Workshops und auch internationale Kongresse ab.

SQS ist seit 2002 auch auf der CeBIT vertreten, und wird auf Webseiten wie www.aboutvb.de (ein deutsches Webmagazin) in höchsten Tönen gelobt. Auch die deutsche Telekom Gruppe lässt nach Angaben von SQS selbst und www.aboutvb.de ihre Software in mehreren europäischen Ländern von SQS testen.

6.3. Government of British Columbia

Genauer gesagt ist es die Homepage des Ministeriums für 'Sustainable Resource Management' des Governments of British Columbia. Diese Seite beschäftigt sich mit Standards für Abnahmetests, die für die Applikationsentwicklung innerhalb des Ministeriums verwendet wird. Darin sind auch ein Template und Instruktionen für die Entwicklung eines Akzeptanztestplans enthalten.

Das Ministerium unterscheidet drei Arten von Akzeptanztests: den **New System Test** – das ist das Testen völlig neu entworfener Software, den **Regression Test** – durchgeführt, wenn ein bereits existierendes Softwaresystem dermaßen geändert wird, dass ein völliges Neutesten stattfinden muss, und der **Limited Test** – auch hierbei besteht das System bereits, die Änderungen sind jedoch so minimal, dass nur die geänderten Features getestet werden müssen; wobei auch das Limited Testen noch in drei Arten unterteilt wird, auf die ich hier aber nicht näher eingehen möchte.

Bezüglich des Test-Kriteriums werden sechs Fälle unterschieden:

1. neue Applikation, die keine existierende ersetzt,
2. neue Applikation, die eine bereits existente ersetzt,
3. Wechsel der zugrunde liegenden Datenbank, ohne Änderung an der Applikation (zB Upgrade von Oracle 7.3 auf 8.05),
4. Erweiterung der bestehenden Applikation mit Funktionalität,
5. Änderung der Funktionalität der bestehenden Applikation, und
6. Änderung der Infrastruktur ohne Änderung der Applikation (zB neue Arbeitsumgebung, neuer Server, oder die Applikation wird auf einem neuen Gebiet eingesetzt).

Zur Entwicklung des Akzeptanztestplans wird ein 12-seitiges Template auf der Homepage zur Verfügung gestellt, dass so umfangreich ist, dass man beim Ausfüllen einfach nichts vergessen kann. Zusätzlich werden auf der Homepage noch detailliert Hinweise, Hilfestellungen und Beispiele zum Ausfüllen der Vorlage gegeben.

6.4. Das WIKI WIKI Web

Bevor ich zum Inhalt dieser Seite komme, möchte ich kurz darauf eingehen wo der Name Wiki eigentlich herkommt, beziehungsweise was das Wiki Web eigentlich ist.

6.4.1. Hintergrund. Das WikiWikiWeb ist eine Datenbank für Entwurfsmuster, die 1995 von Ward Cunningham entwickelt wurde. Es bietet die Möglichkeit ohne HTML-Kenntnisse mit Formularen Text zu editieren. Da das auf diese Weise sehr schnell geht und außerdem auf diese Art eine Datenbank sehr schnell mit Daten angefüllt werden kann, ist der Name Wiki nahe liegend, denn 'Wiki Wiki' bedeutet 'schnell' auf hawaiianisch. Mittlerweile findet man viele solcher Wikis im Internet, die in allen möglichen Bereichen eingesetzt werden, großteils für Enzyklopädien.

(Informationen von http://www.heise.de)

6.4.2. Speziell. Die nun von mir zitierte Wiki Seite beschäftigt sich mit dem Akzeptanztest. Genauer gesagt geht es darum, dass sich Entwickler darüber Gedanken machen, wie sie es schaffen, mit ihren Kunden fehlerfrei zu kommunizieren und die Kunden dazu zu bringen, gute Akzeptanztests zu schreiben. Auch die Frage, ob das für Nicht-Programmierer überhaupt möglich ist, wird hier aufgeworfen, aber noch nicht ganz beantwortet.

Einige auf dieser Seite aufgeworfene Ideen und Vorgehensweisen möchte ich kurz aufzählen:

Ein französischer Codierer, der zwei Bücher und einige Artikel über Extreme Programming schrieb, machte die Akzeptanztests selbst und zwar indem er sich mit den Kunden zusammensetzte und ihnen folgende Frage stellte: "Wenn Sie es wären, der jetzt den manuellen Akzeptanztest machen müsste, was würden Sie verifizieren?" Danach schrieb er den Testcode so, dass nach bestem Wissen und Gewissen die vom Kunden aufgezählten Punkte behandelt wurden.

Die meisten Entwickler im Forum beschränken den Abnahmetest auf einen Test des GU-Interfaces, in der Form: "Der 'Weiter'-Button soll genau an der Stelle (100,100) sein, der 'Zurück'-Button auf (150,100)" oder "Das Backup-Item soll einen Dialog beginnen, der einen Start-Backup-Button hat um das Backup zu beginnen." Allerdings sind solche Tests problematisch, denn wenn das User Interface nicht in dieser Art und Weise spezifiziert ist, wie soll man es dann Testen? Und vor allem wie soll man Tests schreiben, die fehlschlagen wenn das Interface schlampig und schwer zu

verwenden ist, und gut ausfallen, wenn es sauber, einfach und klar ist?

Der meiner Meinung nach interessanteste Beitrag stammt einem pensionierten israelischen Air Force Offizier, der sechs Jahre lang der Kopf des Flugsimulatoren-Softwareentwicklungsteams war und auf diesem Gebiet viel Erfahrung mit Akzeptanztests gesammelt hat. Bei ihm war es so, dass die meisten Kunden den Entwickler die Abnahmetests schreiben ließen, denn die Spezifikationen des Kunden auf diesem Gebiet sind oft recht kurz, wie etwa: "Ich brauche einen Flugsimulator um Notfallprozeduren und Instrumentkampfregeln für dieses Flugzeug zu trainieren". Der Auftragnehmer muss dann die Anforderungen selbst analysieren und aufschreiben. Es gibt zwar einen Formalen Vertrag mit dem Kunden, aber der Entwickler agiert selbst als Kunde und lenkt das Projekt in die Richtung, die seiner Meinung nach eingeschlagen werden sollte. Er schreibt die Geschichten, schreibt die Akzeptanztests, priorisiert, usw. Der Kunde bekommt die fertigen Prozeduren vorgelegt, stimmt zu und sieht bei der Ausführung der Tests zu. Diese Tests dauern 1 bis 3 Wochen. Zusätzlich bekommen die Kunden einige Tage 'Free Flight', in denen sie selbst noch die Simulatoren ausprobieren (lassen) können. Denn das größte Problem dabei ist, dass die Auftraggeber (Bodenpersonal, dass das Geld zahlt, wie etwa American Airlines oder die Chinesische Air Force) hier nicht gleich den Endbenutzern (Piloten) sind. Somit kann es passieren dass ein Pilot des Kunden einmal sagt: "Das fliegt sich nicht gut". Und das ist ein Problem, obwohl es immer noch der Auftraggeber ist, der im Endeffekt das System akzeptiert oder nicht.

7. Abnahmetests und Qualität

In diesem Kapitel versuche ich einen Zusammenhang zwischen dem Abnahmetest selbst und der Qualität des gesamten Softwareproduktes herzustellen. Dazu werde ich als erstes auf acht Qualitätsmaßstäbe eingehen, die H. Sneed in seinem Buch 'Software Qualitätssicherung' [16] als 'messbar' qualifiziert, und diese in Bezug mit dem Akzeptanztest setzen.

Zu den Qualitätsmaßen lt. [16] gehören:

- funktionale Vollständigkeit,
- Benutzerfreundlichkeit,
- Effizienz,
- Zuverlässigkeit,
- Sicherheit,
- Wartbarkeit,
- Erweiterbarkeit und
- Übertragbarkeit

Wie man auf den ersten Blick sehen kann, wenn man Kapitel 2 aufmerksam gelesen hat, stehen die ersten beiden Werte in unmittelbarem Zusammenhang mit dem Akzeptanztest. Denn der Abnahmetest testet ja in erster Linie, ob die Abnahmekriterien erfüllt wurden, und dazu gehört nun einmal die vollständige Funktionalität (Test auf vertragliche Akzeptanz). Und da der Test von den späteren Benutzern durchgeführt wird, kann man sagen, dass somit auch die Benutzerfreundlichkeit getestet wird (Test auf Benutzerakzeptanz), denn wenn das Programm unübersichtlich und schwer zu bedienen ist, wird es von den Endbenutzern wohl nicht akzeptiert werden.

Zur Effizienz finden wir eine Aussage in der Definition am Anfang von Kapitel 1. Denn dort heißt es, der Abnahmetest testet: 'Leistung, Funktion, Effizienz, Performance und Konformität…'.

Auf Wartbarkeit, Erweiterbarkeit und Übertragbarkeit kann der Abnahmetest selbst keinen direkten Einfluss nehmen, da diese eher für den Entwickler selbst relevant sind [16]. Außer die Software soll in verschiedenen Umgebungen eingesetzt werden, denn dann führt man einen Feldtest durch (siehe Kapitel 2.3) und testet somit auch einen gewissen Grad der Übertragbarkeit. Ansonsten gibt es Metriken wie etwa MISRA (für die Automobilindustrie, nähere Informationen: http://www.misra.org.uk) um diese Merkmale zu testen und so die Qualität der Software anzuheben.

Was nun Zuverlässigkeit und Sicherheit betrifft, so sind diese zwei Eigenschaften Gegenstand des Systemstests [3], und wurden somit in der Stufe vor dem Abnahmetest schon getestet.

Der Aufwand für fachliche Tests und Abnahmetests kann zwischen 10% und 30% des Projektaufwandes betragen. Er steigt mit wachsender Komplexität des Projektes [3]. Deshalb ist Testen sehr wichtig. Denn die Erfahrungen zeigen, dass das Risiko eines Fehlschlages reduziert wird und Kosten und Zeit gespart werden, wenn Testen von Projektbeginn an als integraler Bestandteil der Software-Entwicklung geplant und umgesetzt wird [15].

Alles in allem sieht man somit, dass der Akzeptanztest selbst keine geringe Auswirkung auf die Qualität des fertigen Produktes hat. Denn was nützt es, wenn die Software gut wartbar und erweiterbar, sehr zuverlässig und sicher ist, wenn ihr eine wichtige Funktion fehlt? Oder zwar alles vorhanden ist, allerdings so kompliziert zu bedienen ist, dass man ohne Handbuch das Programm nicht einmal beenden, geschweige denn überhaupt starten kann?

Außerdem verbessert es die Qualität des Endproduktes, wenn ein Abnahmetest korrekt durchgeführt

wird, da dadurch Fehler entdeckt werden können, die sonst erst bei der endgültigen Inbetriebnahme entdeckt werden würden. Dadurch wird Zeit und Geld gespart. Und das sind meiner Meinung nach auch zwei wichtige Faktoren, die nicht außer Acht gelassen werden sollten.

8. Konklusion

Abschließend kann man sagen, Abnahmetests sind ein sehr wichtiges Thema, vor allem auch für Softwareentwickler. Denn wie schon eingangs erwähnt, definiert der Abnahmetest den Übergang zwischen Implementierungs- und Wartungsphase [2].

Außerdem sollte der Verkäufer dem Käufer Hilfestellung in der Ausführung geben [11], [10], und da wäre es als Auftragnehmer gut zu wissen, was zu tun ist, oder wo man Informationen darüber findet.

Ich denke, ob und vor allem wie Akzeptanztests wirklich ausgeführt werden, hängt in erster Linie von der Größe des Projektes ab. Bei sehr kleinen Projekten, wie etwa der Erstellung einer Kundendatenbank für einen Kleinbetrieb arbeitet der Entwickler sehr stark mit dem Benutzer zusammen. Dadurch kann der Benutzer laufend prüfen inwiefern seine Anforderungen bereits erfüllt sind, und was noch fehlt. Also entspricht hier der Abnahmetest eigentlich der Zusammenarbeit zwischen Entwicklerteam und Auftraggeber und der ganze Prozess kann eigentlich als Extreme Programming aufgefasst werden. Beispiele dafür wären auch einige 4-h-Praktika an der Uni. Dadurch dass es hier regelmäßige Meilensteine gibt, die einzuhalten sind, und bei denen jeweils auch sämtliche Anforderungen geprüft werden, gibt es hier keinen expliziten Akzeptanztest. Vielmehr wird vom Projektbetreuer zum Schluss zwar alles noch einmal überprüft, aber ohne anfangs vereinbarten Testplan.

Handelt es sich jedoch um ein Großprojekt mit vielen Benutzern, einem eigenen Akzeptanztestmanager, usw.… dann gibt es kein Entkommen. Hier ist ein korrekt durchgeführter Akzeptanztest unumgänglich um ein ordnungsgemäßes, den Anforderungen entsprechendes und qualitativ hochwertiges Endprodukt zu erzeugen.

9. References

[1] Geoinformatik-Lexikon der Universität Rostock, URL: http://www.geoinformatik.uni-rostock.de

[2] Georg Erwin Thaller, *Software-Qualität – Entwicklung, Test, Sicherung*, Sybex-Verlag GmbH, Düsseldorf, 1990

[3] Manfred Rätzmann, *Software-Testing*, Galileo Press GmbH, Bonn, 2003

[4] Dorothy Graham, "Requirements and Testing: Seven Missing-Link Myths", IEEE Software, September/October, 2002, pp: 15-17

[5] Homepage des Ministeriums für 'Sustainable Resource Management' des Governments of British Columbia, URL: http://srmwww.tov.bc.ca

[6] Andreas Spillner, Tilo Linz, *Basiswissen Softwaretest, Aus- und Weiterbildung zum Certified-Tester*, dpunkt-Verlag, Heidelberg, 2003

[7] Martin Pol, Tim Koomen and Andreas Spillner, *Management und Optimierung des Testprozesses – ein praktischer Leitfaden für erfolgreiches Testen von Software mit TPI und TMap*, 2. aktualisierte Auflage, dpunkt.verlag GmbH, Heidelberg, 2002

[8] Norman Parrington, Marc Roper, *Software Test – Ziele, Anwendungen, Methoden*, McGraw-Hill Book Company GmbH, Hamburg, 1990

[9] Edward Kit, *Software Testing in the real world*, by the ACM Press, a division of the Association for Computing Machinery, Inc. (ACM), 1995

[10] Raymond Kehoe, Alka Jarvis, *ISO 9000-3 – A Tool for Software Product and Process Improvement*, Springer Verlag, New York, 1996

[11] William Perry, *Effective methods for Software Testing*, John Wiley & Sons Inc., Canada, 1995

[12] Das Wiki Wiki Web, Internetforum, URL: http://c2.com/cgi/wiki?AcceptanceTest

[13] Ernest Wallmüller, *Software-Qualitätsmanagement in der Praxis, Software-Qualität durch Führung und Verbesserung von Software-Prozessen*, Carl Hanser Verlag München Wien, 2001

[14] Homepage der TÜV Informationstechnik GmbH, URL: http://www.tuvit.de

[15] Homepage der Software Quality Systems AG, URL: http://www.sqs.de

[16] Harry M. Sneed, *Software Qualitätssicherung*, Verlagsgesellschaft Rudolf Müller GmbH, Köln, 1988

[17] Homepage über Extreme Programming, URL: http://www.xprogramming.com

Die Formale Inspektion – eine spezielle Review Technik

Ursula Dittrich
0060922
uditttric@edu.uni-klu.ac.at

Abstract

Hohe Entwicklungskosten, mindere Produktqualität und Fehler im Code sind nur einige der Probleme, die bei der Entwicklung eines Software Produktes auftreten können. Erfahrungsgemäß schleichen sich in jedem Projekt während aller Phasen des Entwicklungsprozesses Fehler ein. Diese können zu erheblichen Aufwänden für unproduktive Nachbearbeitungstätigkeiten führen, sofern sie nicht frühzeitig erkannt und behoben werden. Die Bandbreite reicht vom einfachen Korrigieren eines Tippfehlers über Anpassungen im Design bis zur umfassenden Neukonzeption des Projekts.

Einen wichtigen Beitrag zur frühzeitigen Fehlererkennung leisten auf Projektebene Reviews und Inspektionen. Diese Ansätze können im Gegensatz zum Testen schon eingesetzt werden, sobald ein Zwischenprodukt existiert, auch wenn dieses nicht lauffähig ist. Doch sie erhöhen nicht nur die Softwarequalität, sondern steigern auch die Produktivität von Softwareprojekten. Eine ganz spezielle Variante des Reviews ist die Formale Inspektion, deren Hauptqualitäten eine erhebliche Steigerung der Zuverlässigkeit, der Verwendbarkeit und die Reduktion der Wartungskosten eines Softwareproduktes sind.

1. Einleitung

Mit der zunehmenden Verbreitung von Computern und Mikroprozessoren und steigender Komplexität der Produkte stiegen die quantitativen und qualitativen Anforderungen an Software, wobei die Prozesse um diese Software zu erzeugen immer noch dieselben waren. Dies führte meistens zu wenig befriedigenden Ergebnissen und zu Projektfehlschlägen, wie sehr hohe Kosten, zu lange Entwicklungsdauer und äußerst fehleranfällige Produkte. Daher entstand ein Bedarf an Mechanismen um den Erfolg von Projekten zu gewährleisten. Eine Möglichkeit die Qualität eines Produktes zu steigern ist die Durchführung einer Formalen Software Inspektion. Diese Review Art wurde in den 70er Jahren von Michael Fagan entwickelt und ist deshalb auch unter der Bezeichnung Fagan Inspektion bekannt. Sie ist die ursprüngliche Form der Inspektion und bildet die Basis für die meisten anderen Inspektionstechniken.

Als Fagan 1976 seine Review Technik veröffentlichte, hatte er damit die kosteneffektivste manuelle Qualitätssicherungstechnik geschaffen [Fagan, 1976]. Auch heute, über 25 Jahre später, setzt ein Großteil der führenden Softwareunternehmen diese Technik ein. Selbstverständlich sind computerunterstützte Qualitätstechniken wie unter anderem Modultests oder statische Analysetools unverzichtbar, aber nach wie vor sind die Kollegen eines Softwareentwicklers die idealen Prüfer, wenn es darum geht, Fehler in Dokumenten oder Programmen zu finden.

In dieser Arbeit werde ich den Einsatz der Formalen Inspektion, als Mittel der Qualitätssicherung für Softwareprodukte, genauer betrachten.

Um die Grundlagen einer Inspektion auch verständlich zu machen, werde ich zuerst kurz das Review allgemein erklären, wobei ich insbesondere dessen Ziele und Ergebnisse, den Unterschied zwischen einem Review und einem Test, und die verschieden Review Arten behandeln werde. Nachdem die grundlegenden Dinge erwähnt wurden folgt des Hauptthema: Die Formale Inspektion. Zu Beginn dieses Abschnittes werde ich den Unterschied zwischen den vier existierenden Inspektionsarten erklären. Danach vertiefe ich mich in das Thema der Formalen Inspektion wobei ich mit der Zusammensetzung eines Inspektionsteams und deren genauem Ablauf beginnen werde, um dem Leser eine Vorstellung geben zu können, wie eine solche Inspektion überhaupt vor sich geht. Weiters werde ich darauf eingehen, wie die Fehler, die durch eine Inspektion aufgedeckt werden, kategorisiert werden, was man unter der optimale Inspektionsrate versteht und welche verschiedenen Lesetechniken bei der

Durchführung einer Inspektion eingesetzt werden können. Die Frage, ob der Einsatz dieser Qualitätssicherungsmethode überhaupt wirtschaftlich ist, werde ich in dem darauf folgenden Kapitel beantworten und dann auf die Vor- und Nachteile der Inspektion eingehen. Daraufhin werde ich kurz das Lotus Inspection Data System vorstellen, das eine Möglichkeit bietet die Ergebnisse einer Inspektion zu erfassen, zu archivieren und zu analysieren. Abschließen werde ich meine Arbeit mit einer kurzen Zusammenfassung der behandelten Themen und einer persönlichen Schlussfolgerung beenden.

2. Review allgemein

IEEE definiert ein Review als "ein formelles Treffen, bei dem ein Produkt oder Dokument dem Benutzer, Kunden oder anderen interessierten Personen vorgelegt wird, um es zu kommentieren und abzusegnen" [IEEE, 1990].

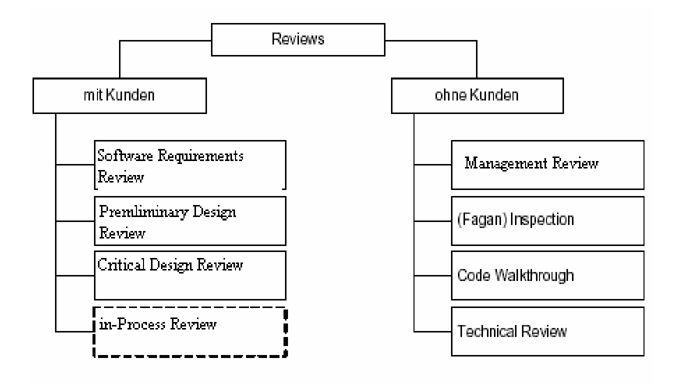


Abbildung 1: Arten von Reviews [Thaller, 2000]

Je nach Anwendungsbereich existieren zahlreiche verschiedene Ansätze für Reviews, die mit bzw. ohne Kunden stattfinden und in definierten Phasen zum Einsatz kommen.

Reviews sind Tätigkeiten, die innerhalb eines Projektteams angewandt werden, um Fehler zu finden. Das Ziel ist neben der eigentlichen Fehlerfindung der Zeitpunkt der Fehlerfindung. Je früher ein Problem erkannt und korrigiert wird, desto geringer sind die Kosten für dessen Korrektur. Beispielsweise kann ein schwerer Fehler im Anforderungsdokument, wenn er erst in der Integrationsphase erkannt wird, sehr hohe Kosten oder im ungünstigsten Fall das komplette Redesign des Produktes zur Folge haben. Das primäre Ziel ist es also, diese Fehler möglichst frühzeitig zu erkennen und zu beheben.

Reviews sind Methoden der Qualitätssicherung, die während des gesamten Entwicklungsprozesses zum Einsatz kommen. Sie beschränken sich in diesem Prozess nicht auf einen bestimmten Abschnitt, daher können sie bereits in der Analyse- und Designphase, als auch während der Entwurfs- und Implementierungphase zur Fehlerfindung eingesetzt werden. Außerdem sind Reviews nicht nur auf die Softwareentwicklung beschränkt, im Gegenteil, sie haben sich bereits in nahezu allen technischen Prozessen bewährt.

2.1 Ziele und Ergebnisse eines Reviews

Ein Review hat das generelle Ziel, Probleme in den Artefakten des Software Prozesses aufzudecken. Es geht also den Fragen nach:

· Erfüllt das geprüfte Ergebnis seinen Zweck?
· Wird das Produkt den Unterlagen entsprechen?
· Sind die Merkmale des Produkts die Geforderten?

Die Hauptziele eines Reviews stellen das Entdecken von Fehlern, das Eingrenzen von Folgekosten, die Behebung von Qualitäts-schwankungen und das Überprüfen, ob das Dokument sich an gewisse Standards hält da. Nebenbei hat ein Review aber auch eine ganze Reihe positiver Nebeneffekte wie zum Beispiel das Einführen und Ausbilden junger Mitarbeiter, die durch das Begutachten der Arbeit anderer einiges hinzulernen können, Verbesserung des Verständnisses des Projekts und die Erzwingung von Sorgfalt bei der Erstellung des Produkts.

Das Ziel eines Reviews ist es, Fehler aufzudecken, nicht sie zu beheben!

Das Ergebnis eines Reviews ist eine Diagnose, die in einem Protokoll festgehalten wird. Dieses gibt wieder, welche Teile als gut befunden werden und nicht geändert werden dürfen, und begründet welche mangelhaft sind und warum diese geändert werden müssen.

2.2 Abgrenzung zwischen Software Test und Review

Reviews und Testen sind sich ergänzende Methoden. Beide haben ihre Stärken und beide sollten in der Softwareentwicklung zur Anwendung gelangen. Ein Unterschied der Beiden Qualitätssicherungsmaßnahmen liegt darin, dass die Formale Inspektion im Gegensatz zum Testen in jeder Phase des Softwareentwicklungsprozesses eingesetzt werden kann. Daher Fehler frühzeitig eliminiert werden wodurch das Auftreten von Folgefehler

vermieden wird, während das Testen der Softwarekomponenten erst dann möglich ist, wenn die Entwicklung des Systems entsprechend weit fortgeschritten ist und zumindest ein Codefragment vorliegt. Ein weiterer Unterschied liegt darin, dass bei der Durchführung eines Reviews viele Fehler auf einmal aufgefunden werden können. Im Gegensatz dazu ist es bei der Durchführung eines Testes häufig so, dass z.B. beim Test eines Codefragmentes der Code nach einem bestimmten Kriterium getestet wird, ein Fehler gefunden wird, dieser korrigiert wird und daraufhin der gesamte Code wieder von vorne gestartet wird um weitere Tests ausführen zu können. Dies erfordert oftmals einen weitaus höheren Zeitaufwand als die Durchführung eines Reviews.

Dennoch ist es in der Praxis häufig der Fall, dass unter Zeitdruck zuerst die Verifikation und Validierung[1] des Produktes vernachlässigt wird, was natürlich zu einer Verminderung der Qualität führt. Ebenso wird die Wichtigkeit und die praktische Relevanz des Themas bei der Ausbildung von Softwareingenieuren oft unterschätzt. Es wird viel Zeit dafür aufgewendet, wie man Systeme und Produkte plant, modelliert und implementiert, und dies wird oft auch fleißig geübt. Wie man aber eine gewisse Qualität in diesen Prozess einbringt, darauf wird nur kurz und meist nur theoretisch eingegangen.

Eine optimale Qualitätssicherung eines Softwareproduktes ist allerdings erst dann gegeben, wenn sowohl Reviews als auch Tests durchgeführt werden. Denn nicht alle enthaltenen Fehler sind mit Hilfe eines Reviews identifizierbar.

Reviews können das Testen zwar ergänzen, aber es niemals ersetzen!

2.3 Abgrenzung der verschiedenen Review Arten

Wie bereits im zuvor erwähnt gibt es einerseits Reviews in Zusammenarbeit mit dem Kunden, andererseits auch verschiedene Arten von internen Reviews bei denen der Kunde nicht mit einbezogen wird. Da die Inspektion, die mein Hauptthema darstellt, zu der Gruppe der Reviews zählt, die den Kunden nicht mit einbeziehen, werde ich nur diese ausführlicher behandeln.

Management-Reviews dienen der formellen Bewertung des Projektplans oder auch um zu überprüfen, inwieweit der Projektstatus dem Projektplan folgt, wobei entschieden wird, ob und wie etwas durchgeführt wird. Sie werden üblicherweise im Projektplan zu bestimmten Zeitpunkten des Software-Life-Cycle festgelegt, beispielsweise während der Analysephase, der Entwurfsphase.

Technische Reviews werden eingesetzt, um einen konkreten Bestandteil der Software zu bewerten, die Übereinstimmung von Spezifikation oder Standards mit dem Softwareprodukt selbst zu überprüfen und auch um Fehler zu finden.

Inspektionen weisen eine hohe Ähnlichkeit zu Technischen Reviews auf. Auch sie werden verwendet, um Fehler in frühen Stadien der Softwareentwicklung zu finden und ein Softwareprodukt im Hinblick auf die Einhaltung von Standards und Vorgabedokumenten, wie etwa das Anforderungsdokument, zu überprüfen. Der Ablauf von Inspektionen ist formaler und erlaubt den direkten Vergleich zu Normen und Standards.

Walkthrough Ein Unterschied zur Inspektion liegt darin, dass die Rollenverteilung anders vorgenommen wird. Bei einem Walkthrough stellt der Autor den Review - Teilnehmern ein Dokument vor und arbeitet dieses Schritt für Schritt mit ihnen durch. Im Gegensatz dazu hat der Autor bei einer Inspektion nur die Aufgabe entdeckte Fehler nachträglich zu korrigieren und Fragen zu beantworten. Außerdem unterscheiden sich diese zwei Arten auch noch durch die Art und Weise, wie das Dokument durchgearbeitet wird. Bei einem Walkthrough wird das Dokument genau so wie es vorliegt, von oben nach unten, durchgelesen, wogegen es beim Einsatz einer Inspektion die verschiedensten Lesetechniken gibt, auf die ich noch später zu sprechen kommen werde.

3. Formale Inspektion

Eine Inspektion wird als eine statische Analysemethode, um Qualitätseigenschaften von Softwaredokumenten zu überprüfen definiert.

Sie ist eine ganz spezielle Art und Weise wie ein Review durchgeführt wird, welche Personen in diese Überprüfung miteinbezogen werden und welche Rollen diese übernehmen. Sie dient dazu, die während des Entwicklungsprozesses generierten Dokumente zu analysieren. Zu diesen Dokumenten gehören unter anderem die Produktanforderungen, Design Diagramme und der Quellcode. Ebenso wie ein Review wird die Inspektion verwendet, um Fehler in frühen Stadien der Softwareentwicklung zu finden und ein Softwareprodukt im Hinblick auf die

[1] V&V: Die ständige Kontrolle während der Entwicklung von Software, um sicherzustellen, dass das Produkt seinen Spezifikationen entspricht und auch korrekt funktioniert

Einhaltung von Standards und Vorgabedokumente, wie etwa das Anforderungsdokument, zu überprüfen.

Da im Laufe einer Software Inspektion das zu entwickelnde System nicht ausgeführt werden muss, wird diese auch als *statische Validierungs und Verifikations Methode* bezeichnet. Diese Review Technik weist eine hohe Ähnlichkeit zum Technischen Review auf. Der Ablauf von Inspektionen ist allerdings formaler und erlaubt dadurch den direkten Vergleich zu Normen und Standards. Aufgrund des hohen Formalismus und relativ geringer Freiheitsgrade bei der Durchführung von Inspektionen, ist diese Methode auch für Schulungszwecke innerhalb des Projektteams gut einsetzbar, da sowohl der primäre Nutzen von Inspektionen – die Fehlerfindung – als auch das Kennen lernen des Produkts und der Methode unterstützt wird.

Inspektionen werden nach genau vordefinierten Schritten durchgeführt. Dabei sind sowohl die jeweiligen Phasen, als auch die zu ermittelnden Messdaten als auch die jeweiligen Rollen genau vordefiniert. Inspektionen weisen bezüglich Prozessablauf und der Teilnehmerzahl einen höheren Detaillierungsgrad auf. Diese Aufteilung kann jedoch ebenfalls für Reviews eingesetzt werden.

Bei Inspektionen wird ein Softwaredokument z.B. Anforderungen, Konzepte, Szenarien, Design, Code) von einem oder mehreren Inspektoren auf Fehler oder Mängel hin untersucht. Software Inspektionen können in jeder Phase des Softwareentwicklungszyklusses eingesetzt werden. Im Vergleich zu dynamischen Verfahren, wie zum Beispiel dem Testen, können Inspektionen aber schon sehr viel früher in der Entwicklung eingesetzt werden. Dadurch werden die Qualitätsdefizite dort gefunden und beseitigt, wo sie auch tatsächlich entstehen. Dies führt zu einer erheblichen Reduzierung der anfallenden Fehlerkosten.

3.1 Arten von Inspektionen

Die *Fagan/Formale Inspection* ist die ursprüngliche Form einer Inspektion und bildet die Basis für die meisten anderen Inspektionstechniken.

Das *Active Design Review* konzentriert sich, wie der Name schon sagt, hauptsächlich auf Inspektionen eines Software- Designs.

Die *N-Fold Inspektion* zielt darauf ab, eine Anzahl *n* kleiner Inspektionsteams einzusetzen, die jeweils der Methode von Fagan folgen. Der Grundgedanke geht davon aus, dass kleine Gruppen höhere Fehlerfindungsraten aufweisen als vergleichsweise große Gruppen. Diese Methode wird von seinen Erfindern Martin und Tsai hauptsächlich für besonders kritische Projekte vorgeschlagen, nachdem sie relativ kostenintensiv ist [Martin, Tsai, 1990].

Eine Mischform aus den bisher genannten Inspektionsarten ist die so genannte *Phased Inspektion*, die von Knight und Myers entwickelt wurde [Knight, 1993]. Dabei wird das Artefakt in mehreren Teilinspektionen untersucht, die Phasen genannt werden. Eine normale Inspektion kann bis zu 6 Phasen haben, wobei jede Phase ein bestimmtes Ziel verfolgt. Die Inspektoren untersuchen das Artefakt nur im Hinblick auf das jeweilige Ziel. Solange das aktuelle Ziel nicht erreicht ist, daher noch nicht alle Korrekturen erledigt sind, wird nicht zur nächsten Phase weitergegangen.

3.2 Inspektionsteam

An der Inspektion nehmen mehrere Personen teil. Die richtige Anzahl der Teilnehmer ist wichtig für die Effizienz einer Inspektion. Zu viele Teilnehmer erhöhen zwar den Aufwand, bringen aber nur geringfügig mehr Nutzen. Idealerweise sollte die Gruppe aus mindestens 3 aber höchstens 6 Personen bestehen.

Nun stellt sich noch die Frage welche Rollen diese Personen übernehmen:

- Der *Moderator*: Er ist für die Planung, den Ablauf, die Organisation und die Durchführung der Inspektion verantwortlich. Er hat insbesondere dafür zu sorgen, dass die Sitzungen effizient und in geordneten Bahnen stattfinden. Er ist sowohl für die Auswahl der Teilnehmer als auch für die Bereitstellung der benötigten Informationen und Unterlagen, die aus Sourcecode, Dokumentationen, Produkt Dokumenten, Anforderungsanalysen, Spezifikationen, Header Files, Regeln und Checklisten bestehen, zuständig.
- Der *Autor* ist der Urheber des zu inspizierenden Objektes oder ein Repräsentant des Teams, das den Prüfling erstellt hat. Er hat die Aufgabe auftretende Fragen zu klären und Hintergrundinformationen bereitzustellen, allerdings ist es ihm nicht erlaubt aktiv an der Inspektion teilzunehmen oder gar Lösungen vorzuschlagen oder zu rechtfertigen. Weiters fällt ihm das spätere Korrigieren der bei der Inspektion aufgedeckten Fehler zu.
- Der *Gutachter* geht das Dokument im Laufe der Inspektion zeilenweise durch und deckt Fehler und Verstöße gegen Programmierstandards auf. Er hat generell die Aufgabe, sich anhand der vom

Moderator weitergeleiteten Unterlagen gut auf die Sitzungen vorzubereiten. Weiters muss er sich natürlich aktiv daran beteiligen.

- Der *Protokollführer* übernimmt die Rolle des Berichterstatters über den Verlauf der Sitzungen. Er protokolliert alle Ergebnisse bzw. gefundenen Fehler mit und erstellt daraus einen Inspektionsbericht, der den Teilnehmern später zur nochmaligen Überprüfung zur Verfügung gestellt wird. Dieses Dokument dient auch als Grundlage der Fehlerkorrektur.

Ein ideales Team besteht aus je einem Moderator, Autor und Protokollführer und drei Gutachtern. Eine Person kann allerdings auch mehrere Rollen übernehmen. Der Moderator und der Protokollführer können zum Beispiel gleichzeitig auch Gutachter sein. Der Autor darf allerdings keine weitere Rolle übernehmen, da er voreingenommen ist. Aus diesem Grund ist es auch möglich, dass die Idealbesetzung, die, wie schon erwähnt, aus je einem Moderator, Autor und Protokollführer und drei Gutachtern besteht, von nur drei Personen übernommen wird.

3.3 Ablauf

Thaller beschreibt Fagans Inspektionsansatz in folgenden 6 Schritten [Thaller, 2000] :

- *Planungsphase (Planning Phase)*: Nach der Fertigstellung eines Dokumentes wird ein Inspektionsteam ernannt. Ein Teammitglied übernimmt die Rolle des Moderators und ist verantwortlich für den organisatorischen Ablauf.
- *Einführung (Tutorial)*: Neue Teammitglieder oder Teams, die mit der Methode nicht vertraut sind, lernen die Grundregeln und Techniken der Inspektion. Diese optionale Einführung kann einerseits zum Verständnis der Methode und andererseits zum besseren Verständnis der Anwendungsdomäne oder des Inspektionsartefakts eingesetzt werden.
- *Vorbereitungsphase (Preparation Phase)*: In einem Zeitraum von ungefähr zwei Stunden (bei hoher Komplexität kann die Vorbereitungsphase umfangreicher sein) bereiten die Inspektoren die nötigen Hilfsmittel vor, die sie später für die Inspektion brauchen. Das primäre Ziel dieser Phase ist die Einarbeitung in den Anwendungsbereich und die Auseinandersetzung mit Inspektionen.
- *Ausführung (Operation)*: Unter der Leitung des Moderators inspiziert das Team das Softwareprodukt. Auch bei der Inspektion ist die Fehlerfindung das Hauptziel. Alle Fehler werden dokumentiert und dabei nach Typ, Schwere, Auswirkung, usw. katalogisiert. Dabei geht es nicht um mögliche Verbesserungen oder Korrekturen, sondern nur um die Erfassung der Fehler. Die subjektive Entscheidung über die Verwendbarkeit des Dokumentes, d.h. ob eine Überarbeitung notwendig ist oder das Dokument freigegeben werden kann, beendet die Ausführungsphase.
- *Überarbeitung (Rework)*: Basierend auf den notierten Mängeln überarbeitet der Autor das inspizierte Dokument.
- *Überprüfung der Richtigkeit (Verification)*: Nachdem alle Mängel behoben worden sind, überprüft der Moderator die Modifikationen und kann gegebenenfalls einen zweiten Inspektionszyklus (die Re-Inspektion) initiieren.

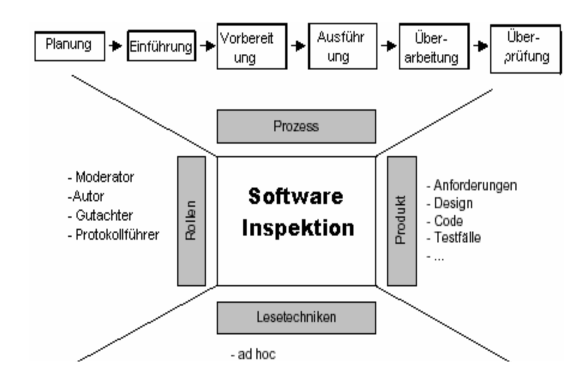


Abbildung 2: Das Zusammenspiel der einzelnen Elemente einer Inspektion

3.4 Fehlerarten

Die während einer Inspektion identifizierten Fehler werden grundsätzlich in zwei Klassen unterteilt: in Major beziehungsweise Minor Defekte.

Als Major Defekte werden die Fehler bezeichnet, die möglicherweise erheblich höhere Kosten verursachen, wenn sie zu einem späteren Entwicklungszeitpunkt gefunden werden. Im Gegensatz dazu bleibt der Aufwand zur Behebung eines Minor Defektes immer derselbe. Es hat dementsprechend keine größeren Auswirkungen auf den Zeit- oder Kostenplan, ob der Fehler sofort oder erst später korrigiert wird. Das bedeutet allerdings nicht, dass der Defekt nicht behoben werden muss. Allerdings ist es möglich, dass sich ein Minor Defekt, wenn er nicht rechtzeitig behoben wird, im Laufe des

Softwareentwicklungsprozesses in einen Major Defekt verwandelt.

Die Inspektion konzentriert sich hauptsächlich auf die Auffindung der Major Defekte.

3.5 Optimale Inspektionsrate

Inspektionen sind dann am erfolgreichsten, wenn die Inspektoren gründlich vorbereitet zur Sitzung kommen. Ca. 80% der Fehler, die durch Reviews entdeckt werden können, finden die Gutachter schon in der Vorbereitungsphase! In der eigentlichen Reviewsitzung werden also (nur) noch die restlichen 20% der Fehler gefunden. Dies zeigt, dass eine gründliche Vorbereitungsphase ein kritischer Erfolgsfaktor ist.

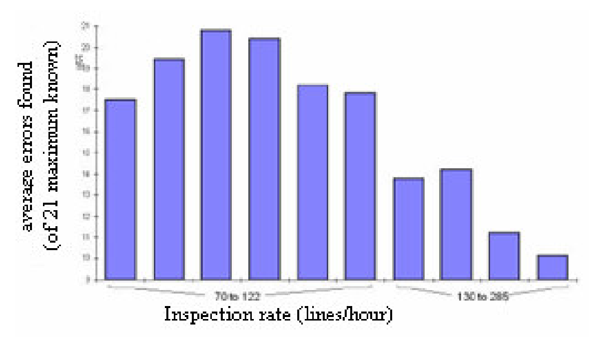


Abbildung 3: Die optimale Inspektionsrate [SETE, 2003]

Es gibt Erfahrungswerte, wie viel Zeit sich ein Reviewer für die Vorbereitung nehmen muss, um die oben erwähnten 80% der Fehler finden zu können. Genau diese Erfahrungswerte dienen als Grundlage für die optimale Inspektionsrate, die für Programme in der Regel eine Stunde für 100 - 150 NLOCs (non-commentary lines of code) vorsieht. [SETE, 2003] Dies gilt ziemlich unabhängig von der Programmiersprache.

Für Textdokumente ist diese Prüfgeschwindigkeit stärker abhängig von der Art des zu prüfenden Dokuments. Sie beträgt oft nur einige wenige Seiten pro Stunde und liegt bei extrem kritischen Dokumenten wie zum Beispiel bei den Anforderungsdefinitionen für ein neues Flugzeug oder anderen technischen Produkten bei einer Seite pro Stunde oder sogar noch darunter!

3.6 Lesetechniken

Um die Effizienz eines Inspektionsteams zu verbessern, wurden Methoden entwickelt, die den Inspektionsteilnehmern den Umgang mit Dokumenten erleichtern und den Fehlerfindungsprozess unterstützen [Visek, 2003]. Dieses Kapitel beschäftigt sich mit einem Ansatz zur Fehlersuche, den Lesetechniken, die den Inspektor bei der Arbeit unterstützen. Diese Dokumente und Fragenkataloge werden den Inspektoren in der Einführungsphase zusammen mit den anderen Dokumenten vom Moderator übergeben.

Die einfachste aller Lesetechniken ist die *ad-hoc* Methode, welche die erste aller Untersuchungsmethoden war und dank Fagan schnell verbreitet wurde. Darunter versteht man ein lineares Abarbeiten bzw. Lesen des jeweiligen Dokumentes. Diese Technik kann in einer Firma ohne grosse Anpassungen und Voraussetzungen eingesetzt werden, da ohne bestimmte Leseanleitung einfach nach Fehlern gesucht wird. Das ad-hoc Verfahren kann zwar auf jedes Dokument angewendet werden, kann aber aufgrund der unterschiedlichen Zugänge der Inspektoren und der fehlenden Anleitungen, gegebenenfalls nicht gleichartig wiederholt werden. Sie wird zwar in der Praxis immer weniger eingesetzt, liefert aber Vergleichswerte bei der Untersuchung der Effektivität und Effizienz verschiedener Lesetechniken

Werden *Checklisten* basierte Lesetechniken angewandt, so wird eine Liste, bestehend aus einer Reihe von Fragen, die sich einerseits auf allgemein gültige Dinge, wie etwa Vollständigkeit, Konsistenz oder Layout, und andererseits auf spezifische, dem Anwendungsgebiet angepasste Inhalte beziehen, erstellt. Die Inspektoren, also die lesenden Teammitglieder, untersuchen den Prüfling bezüglich der vorgegebenen Kriterien der Liste. Diese Kriterien, die jeweils auf die Anwendungsdomäne und die konkrete Dokumentart abgestimmt werden müssen, werden als Richtwerte für ein gutes Dokument herangezogen. Dieser Fragenkatalog wird allen teilnehmenden Inspektoren ausgehändigt. Die Inspektoren lesen das Dokument im Hinblick auf diese Punkte durch. Ihre Aufgabe ist es, diese Fragen vor, während und nach dem Lesen des Software-dokumentes zu beantworten und entsprechend zu dokumentieren. Die Prüfung des Artefakts ist beendet, wenn alle Checklistenpunkte bearbeitet wurden, keine Fehler mehr gefunden werden. Wurde der vordefinierte Zeitrahmen erreicht bzw. gesprengt, dann wird die Prüfung zwar ebenfalls abgebrochen, um die Gutachter nicht zu überfordern, aber zu einem späteren Zeitpunkt wieder aufgenommen. Checklisten sind besonders gut bei gut bekannten Anwendungsdomänen oder bei ähnlichen Inspektions-Objekten anwendbar, da bereits auf Erfahrungen

zurückgegriffen werden kann und eventuell schon existierende Checklisten wieder verwendet werden können. Der grundlegende Vorteil der Checklisten basierten Lesetechnik besteht darin, dass sie eine bessere Verteilung der Arbeit ermöglicht. Trotzdem ist sie wegen der allgemeinen Fragen und dem Mangel an konkreter Untersuchungsstrategie unsystematisch.

Die *Szenario* basierte Lesetechnik basiert auf vordefinierten Rollen innerhalb des Entwicklungs- beziehungsweise des Inspektionsteams. Es werden verschiedene Sichtweisen, Fehlerklassen und Dokumententeile in den Mittelpunkt der Untersuchung gestellt. Dabei werden den Inspektoren in konkreten Anleitungen (den Szenarien) sowohl die Vorgangsweise, als auch die wichtigen, zu beachtenden, Kriterien vorgegeben. Die Inspektoren versetzen sich also in die Lage einer Rolle und untersuchen das Dokument hinsichtlich der entsprechenden, für die jeweilige Rolle definierten Aspekte. Mögliche und in der Praxis häufig verwendete Rollen sind beispielsweise die Designer-, Tester-, Anwenderrolle [Biffl, 2001]. In der Regel gibt es für die jeweiligen Rollen geeignete Leitfäden, in denen die wesentlichen Schritte definiert sind. Aufgrund der rollenspezifischen Vorgangsweise müssen diese Leitfäden bei geänderten Anwendungsdomänen nur geringfügig modifiziert werden. Durch die verschiedenen Sichten der Rollen werden unterschiedliche Fehlerarten aufgefunden. Bei der Anwendung von szenariobasierten Leetechniken ist zu beachten, dass alle Fehlerkategorien durch die Rollen gefunden werden können. Bei dieser Methode muss darauf geachtet werden, dass durch die Anwendung der Lesetechnik auch alle möglichen Fehler tatsächlich gefunden werden können. Die Aufgabenpakete müssen daher entsprechend gestaltet sein.

Die angewendete Lesetechnik bestimmt insbesondere die Effizienz und Effektivität der Inspektion, d.h. wie viele Fehler werden in der Inspektion gefunden und mit welchem Aufwand geschieht dies. Welche dieser drei Lesetechniken eingesetzt werden soll hängt ganz und gar vom zu inspizierenden Dokument ab. Checklisten sind besonders gut bei gut bekannten Anwendungsdomänen oder bei ähnlichen Inspektions-Objekten anwendbar, da bereits auf Erfahrungen zurückgegriffen werden kann und eventuell schon existierende Checklisten wieder verwendet werden können. Die Szenario basierte Lesetechnik sollte dann eingesetzt werden, wenn das zu untersuchende Artefakt verhältnismäßig groß, komplex und unübersichtlich ist. Die ad- hoc Methode wird, wie schon erwähnt, heutzutage kaum mehr eingesetzt, da sie im Vergleich zu den anderen Techniken relativ uneffizient ist.

Tabelle 1: Überblick der Lesetechniken

Lesetechnik	Anwendungsbereich	Vorteile	Nachteile
Checklisten basiert	Traditionelle Methode zur Fehlersuche; derzeit meist "state of the praxis" in Unternehmen	Wenig kompelx; viel Literatur und Erkenntnisse verfügbar; im allgemeinen besser als ad-hoc	fehlende Struktur problematisch; bei jedem Projekt auf neuem Gebiet muß eine neue Checkliste erstellt werden
Szenario basiert	Wird heute in großen Unternehmen und Projekten verwendet z.B. der NASA	Gute Fehlerabdeckung durch unterschiedliche Perspektiven; geringe Überschneidung bei Fehlersuche; viel Literatur und Experimente vorhanden; Fehlerfindungskosten oft geringer als bei Checklisten	Teilweise komplizierter in Vorbereitung und Durchführung als Checklisten;

3.7 Wirtschaftlichkeit

Die Inspektionsaufwände sind abhängig von der Projektdauer, die mit der Verfügbarkeit des Materials beginnt und mit der Korrektur aller gefundenen Fehler endet, und der erbrachten Leistung. Diese Kosten schließen die Trainingskosten des Personals, die Management-kosten, die Vorbereitungs- und Durchführungskosten der Sitzung und zusätzlich die Kosten der Fehlerkorrektur mit ein.
In der Regel nehmen durchschnittlich 4-5 Personen an einer Inspektion teil. Sie benötigen 1-2 Stunden für die Vorbereitungen und weitere 1-2 Stunden für die Sitzung. Bei der Durchführung einer Code Inspektion ermöglicht diese totale Leistung von 10-20 Stunden im Durchschnitt die Entdeckung von 5 bis 10 Fehlern in 250-500 Zeilen eines neuen, ihnen vollkommen unbekannten Codes oder 1000-1500 Zeilen eines normalen, ihnen schon bekannten, aber eventuell ergänzten Codes. [O´Neil, 1995]

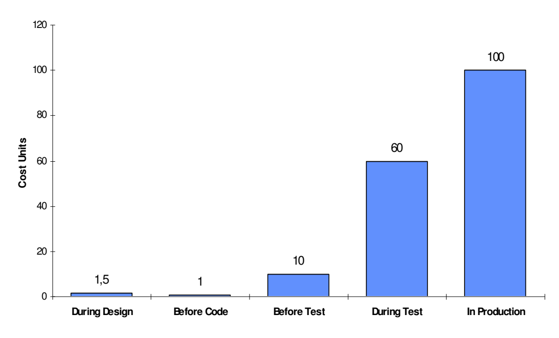


Abbildung 5: Relative Fehlerbehebungskosten

Der Nutzen dieses Prozesses beinhaltet eine Reduktion des Aufwandes für die Neubearbeitungen, Tests und Wartung.

Das Ersparnis ist umso höher, je früher ein Fehler im Softwareentwicklungsprozess gefunden wird. Spätere Entdeckungen können 2 bis 10 Mal teurer sein. [O´Neil, 1995]

3.8 Vor- und Nachteile der Inspektion

Software Inspektionen sind ein wichtiges Instrument, um hohe Qualität im Prozess der Softwareentwicklung zu erreichen. Je früher Fehler entdeckt werden, desto geringer ist der Kosten- und Zeitaufwand, der erbracht werden muss, um sie zu beheben. Wo diese Kosteneinsparungen in einem Softwareprojekt möglich sind, zeigt Abbildung 4 mit den verschiedenen Phasen eines Softwareprojekts. In jeder dieser Phasen gibt es einen Anteil an Tätigkeiten, der nur deshalb nötig ist, weil Menschen nicht von vornherein absolut fehlerfrei arbeiten können (in der Abbildung mit "Rework" bezeichnet). Wenn zum Beispiel im Integrationstest ein Fehler gefunden wird, kann es sein, dass sich dieser Fehler als Designfehler herausstellt. Dann muss ein Designdokument geändert werden, einige Funktionen müssen neu programmiert werden, Modultests müssen nachgeholt werden, und dann erst kann der Integrationstest wieder aufgenommen werden. Dieser ganze Aufwand wäre nicht nötig gewesen, wenn der Fehler bei einer Inspektion des Designdokuments gefunden worden wäre! Durch den regelmäßigen Einsatz von Inspektionen können circa 2/3 des Rework-Anteils eines Softwareprojektes so durch Inspektionen "wegoptimiert" werden.

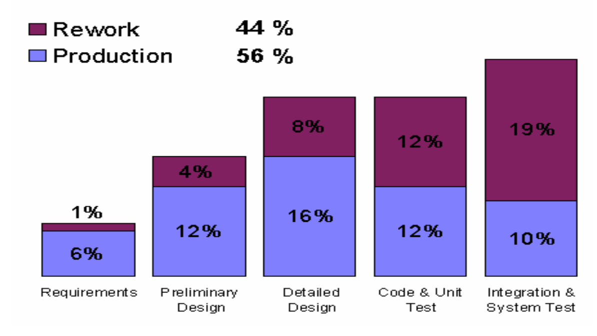


Abbildung 4: Rework-Anteil in einem Software-Projekt [Wheeler, 1996]

Eine effektive Durchführung der Inspektion ermöglicht eine wesentliche Verbesserung der voraussichtlichen Terminplanung, eine substantielle Reduktion der Entwicklungs- und Wartungskosten, eine Erhöhung der Kundenzufriedenheit sowie der Moral des Entwicklungsteams.

Da die Formale Inspektion im Gegensatz zum Testen in jeder Phase des Softwareentwicklungsprozesses eingesetzt werden kann, werden durch ihre Anwendung Defekte dieser Art noch vor Abschluss der jeweiligen Phase erkannt und können eliminiert werden, wodurch das Auftreten von Folgefehler vermieden wird. Wird während der einzelnen Entwicklungsphasen eines Projektes regelmäßig eine Inspektion durchgeführt, so werden bis zur Fertigstellung 2/3 der enthaltenen Defekte erkannt und behoben sein.

Doch nicht nur die finanziellen Aspekte spielen eine wichtige Rolle, auch die Zuverlässigkeit und Verwendbarkeit eines Softwareproduktes wird durch den Einsatz einer Inspektion erheblich gesteigert. Außerdem werden auch allfällige Wartungskosten reduziert. Die Inspektion ist eine sehr effiziente Methode, um Fehler zu vermeiden, da viele Defekte gleichzeitig aufgedeckt werden können, wogegen bei einem Software Test der Code oft nach einem fehlerhaften Resultat abgebrochen und der Fehler zuerst korrigiert werden muss. Führt man eine Inspektion durch, so ist nicht nur eine erhebliche Verbesserung der Softwarequalität zu erwarten, sondern auch eine bessere Produktivität der Projektteams. Neben der Fehlerkorrektur kann in einer Formalen Inspektion auch überprüft werden, ob das Dokument sich an gewisse Standards und Normen, wie zum Beispiel den IEEE Standard, hält.

Ein weiterer Nutzen von Inspektionen zeigt sich, nachdem das Softwareprojekt abgeschlossen ist. Mit dem Auslieferungstermin ist der Software-Lebenszyklus bekanntlich nicht zu Ende, denn in der Wartungsphase wird oft sogar noch mehr Aufwand in die Software investiert als für die Ersterstellung nötig war. Software, die bei der Ersterstellung einem Review unterzogen wurde, verursacht weit geringere Wartungskosten als Software, bei der dies nicht geschah. Die Wartungskosten können durchaus um den Faktor 10 bis 30 niedriger ausfallen!

Dazu ein Beispiel aus England: in der Firma ICI wurden Inspektionen eingeführt und im Laufe der Zeit bei 400 Programmen angewandt. Später wurden die Inspektionen wieder eingestellt, vermutlich, weil man den Nutzen von Inspektionen nicht eingesehen hatte und "Kosten senken" wollte. Ein Jahr danach wurden die Auswirkungen dieser Entscheidung untersucht und dabei festgestellt, dass die 400 Programme, bei denen Inspektionen durchgeführt worden waren, nur ein Zehntel der Wartungskosten verursacht hatten wie

400 vergleichbare Programme ohne Inspektionen [Integrata, 2000].

Das heißt mit anderen Worten, wenn der Zeithorizont der Betrachtung nur bis zum Auslieferungstermin der Software reicht, dann können "nur" die oben erwähnten Produktivitätssteigerungen von 25% - 35% erreicht werden. Wird aber die Wartungsphase mit berücksichtigt, z.B. weil die Software von der eigenen IT-Abteilung gewartet werden muss, dann tragen Inspektionen noch deutlich darüber hinaus zur Kostenreduktion im eigenen Unternehmen bei.

Trotz all dieser Vorteile ist zu erwähnen, dass Inspektionen das Testen zwar ergänzen, aber es niemals ersetzen können! Ein erwähnenswerter Nachteil der Inspektion besteht darin, dass sie keinen oder nur sehr wenig Aufschluss über das Laufzeitverhalten und das Zusammenspiel der einzelnen Komponenten einer Software Auskunft gibt. Außerdem werden wie schon zuvor erwähnt mit einer Inspektion nicht alle der in einem Softwareprojekt enthaltenen Fehler aufgedeckt, die restlichen müssen dann, bevor das Produkt dem Kunden übergeben wird, durch gründliches Software Testen ausgemerzt werden. Das bedeutet zwar möglicherweise einen großen Arbeits- und Zeitaufwand ist aber unumgänglich, wenn das Produkt einem hohen Qualitätsstandard entsprechen soll.

4. Tool unterstützte Erfassung - Lotus Inspection Data System

Eine Möglichkeit die Ergebnisse einer Inspektion zu erfassen, zu archivieren und zu analysieren, stellt das Lotus Inspection Data System dar. Dieses frei verfügbare Tool [Download] hilft die erfassten Ergebnisse einer Inspektion in tabellarische und graphische Form zu bringen. Die so aufbereiteten Daten sind dann auch von Außenstehenden übersichtlicher und leichter zu verstehen.

Die fünf Arbeitsbereiche, in die das Programm unterteilt ist, werden in LIDS, Defekte, Performance, Control und Dispersion bezeichnet. Die Menüs, die Macros, der Datenspeicher und der Protokoll Bereich befinden sich im LIDS Bereich. Die anderen vier Bereiche beinhalten die „Modell“ Charts, die als Grundlage für verschiedene graphische Analysen dienen und ansonsten für die internen Prozesse des Lotus Inspection Data Systems benötigt werden.

Der Benutzer dieses Systems bewegt sich dementsprechend nur im Bereich des LIDS. Dort werden die durch die Inspektion gewonnen Daten eingetragen und mit Hilfe des Tools archiviert und ausgewertet. Das Tool stellt Funktionen zur Verfügung, die dabei behilflich sind die Ergebnisse einer Inspektion wiederum zu reviewen. Weiters hilft es dem Benutzer die dabei erarbeiteten Ergebnisse in tabellarische und graphische Form zu bringen.

LIDS produziert keinerlei druckfertigen Protokolle oder Graphiken, präpariert aber Protokolldaten und kopiert Graphiken in das Windows Clipboard. Diese Dateien können dann in vielfältiger Art und Weise weiterverwendet werden.

Mit Hilfe der, in diesem Tool erstellten Reports, können Graphiken, wie in Abbildung 7, und Tabellen generiert werden.

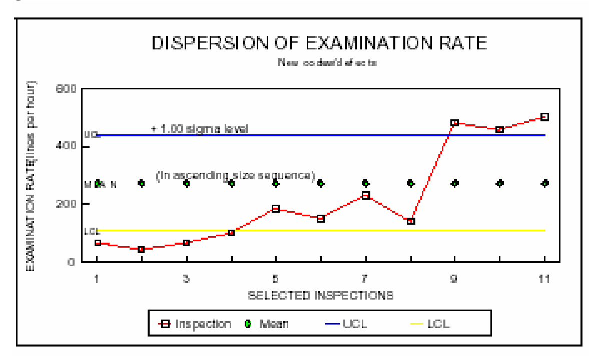


Abbildung 7: Beispiel einer Ausgabegraphik

Leider ist dieses Tool schon etwas veraltert und bietet dem Benutzer im Bereich der Arbeits-oberfläche keinen besonderen Komfort. Außerdem funktioniert LIDS nur mit Lotus 1-2-3 Release 5.01 oder älteren Versionen. Laut Auskunft von Robert Ebenau, dem Autor von LIDS, ist nicht geplant, LIDS für neuere Versionen lauffähig zu machen. Da Lotus 1-2-3 Release 5.01 jetzt nicht mehr im Handel verfügbar ist, muss es anderweitig beschafft werden.

Die Idee einer Toolunterstützung zur Verwaltung der Inspektionsdaten ist zwar grundsätzlich gut und nützlich, entspricht aber in der Version, die mir zur Verfügung stand, in keinster Weise mehr dem heutigen Softwarestandard.

5. Zusammenfassung & Schlussfolgerung

Um qualitativ hochwertige Softwareprodukte herstellen zu können sind zahlreiche Überprüfungen der Produkte und Zwischenprodukte notwendig. Diese Überprüfungen müssen in jeder Phase der Softwareentwicklung umgesetzt werden. Reviews und Inspektionen sind gut geeignet, um diese Überprüfungen bereits in frühen Phasen der Softwareentwicklung für alle Arten von Softwareprodukten, beispielsweise Anforderungs-

dokumente, durchzuführen. Die Hauptgründe dafür, warum gerade in der frühen Entwicklung, also in der Anforderungsanalyse, Planung und Spezifikation, Fehler vermieden werden sollten, liegen darin, dass diese Fehler viel Zeit und Geld kosten können. Das deswegen, weil solche Fehler, wenn sie erst in späteren Phasen des Entwicklungszyklus entdeckt werden (im schlechtesten Fall bei der Übernahme durch den Kunden), sich durch alle vorherigen Phasen ziehen können und damit den Nachbesserungsaufwand oft deutlich vergrößern.

Das Hauptziel von Reviews und Inspektionen ist die Fehlerfindung in Softwareprodukten. Um dieses Ziel zu erreichen, sind Methoden notwendig, die es erlauben, strukturiert und kontrolliert nach Fehlern zu suchen. Lesetechniken sind ein Ansatz, um die Effizienz und Effektivität der Fehlerfindung in Softwaredokumenten zu erhöhen.

Eine Möglichkeit, um die Ergebnisse einer Inspektion zu verwalten, zu archivieren und zu analysieren, stellt LIDS da. Allerdings ist dieses Tool nicht sehr bedienungsfreundlich und basiert auf eine Software, die im Handel nicht mehr erhältlich ist. Die Erstellung eines dementsprechend modernen Tools zur Unterstützung des Reviewprozesses und dessen Auswertung währe daher eine sinnvolle Weiterentwicklung im Bereich der Qualitätssicherung.

6. Referenzen

[Biffl, 2001] Biffl, Stefan: „Software Inspection Techniques to support Project and Quality Management”, Shaker Verlag, 2001

[Download] http://home.att.net/~rge.com/lids/lids.htm?

[Fagan, 1976] Fagan, M.E: „Design and Code Inspections to Reduce Errors in Program Development“, 1976, IBM System Journal

[IEEE, 1990] IEEE Standard Glossary of Software Engineering Terminology, 1990

[Knight, 1993] Knight, J. C. and Myers, E. A. :“An improved Inspection Technique“, in Communications of the ACM

[Martin, Tsai, 1990] Martin, J.; Tsai, W. T.: „N-Fold Inspection: A Requirements Analysis Technique“, in Communications of the ACM

[O´Neil, 1995] Don O’Neill, National Software Quality Experiment: Result 1992-1999, Software Technology Conference, Salt Lake City (1995, 1996, 2000)

[Integrata, 2000] Integrata Training News 4/2000

[SETE, 2003] Elke Hochmüller, Roland Mittermeir, Heinz Pozewaunig, 2002/03, „Software-Entwurf, Test und Entwicklungs-prozess“, Institut für Informatik-Systeme Universität Klagenfurt

[Thaller, 2000] Thaller, Georg Erwin: „*Software Qualitat*“, VDE Verlag, 2000

[Visek, 2003] Visek, 2003, Virtuelles Software-Engineering Kompetenzzentrum; http://www.visek.de

[Wheeler, 1996] Wheeler, 1996, „Software Inspection: an Industry best Practice“

Qualitätssicherung bei agiler Softwareentwicklung

Mario Graschl
9760974
mgraschl@edu.uni-klu.ac.at

Abstract

Die agile Softwareentwicklung ist eine neue Methode innerhalb der Softwareentwicklung. Agile Softwareentwicklung verspricht eine Technik, die sich leicht den geänderten Kundenanforderungen während eines Projektes anpasst. Es stellt sich nun die Frage, ob dies die Art wie Qualitätssicherung betrieben wird beeinflusst. Dieses Papier gibt einen Überblick über die agile Softwareentwicklung und vergleicht sie mit dem traditionellen Ansatz. Weiters wird aufgezeigt wie Qualitätssicherung betrieben wird und in wie weit diese sich beim agilen Ansatz ändert.

1. Einleitung

Agile Softwareentwicklung versucht sich der Herausforderung zu stellen, schnell an geänderte Kundenwünsche zu reagieren und diese schnellstmöglich umzusetzen. Diese Art von Softwareentwicklung hat ihren Ursprung durch das Aufkommen von eXtreme Programming [6] [7] und wird insbesondere durch die steigende Anzahl von Internetprojekten gestützt. Internetprojekte beziehungsweise Softwareentwicklung für das Internet sind Paradebeispiele für das Problem der häufigen Änderungen von Kundenanforderungen während des Verlaufs eines Projektes. Gerade hier versuchen die Methoden der agilen Softwareentwicklung sich zu etablieren und eine Lösung zu bieten. Diese Lösung steht im Gegensatz zu den herkömmlichen Methoden der Softwareentwicklung, die dieses Problem durch ihre strenge Prozessorientiert zu lösen versuchen. Agile Prozesse versuchen durch einen angepassten und beschleunigten Entwicklungsprozesses und durch die Einbindung des Kunden, dieses Problem zu meistern. Dabei stellt sich aber die Frage welchen Platz die Qualitätssicherung hierbei einnimmt und ob dadurch die Qualitätssicherung beeinflusst wird.

Diese Arbeit untersucht diese Frage und untersucht agile Softwareentwicklung hinsichtlich der Qualitätssicherung und ihren Maßnahmen.

2. Qualitätssicherung

Qualitätssicherung wird durch eine Eigenschaft gegenüber anderen Tätigkeiten innerhalb eines Softwareentwicklungsprozesses besonders hervorgehoben – ständige Begleitung der einzelnen Phasen innerhalb des Softwareentwicklungsprozesses. Durch diese Eigenschaft wird der Qualitätssicherung eine besondere Bedeutung zugeordnet, die zu oft vergessen wird. Qualitätssicherung trägt wesentlich zu dem Erfolg eines Softwareentwicklungsprozesses bei.

Qualitätssicherung umfasst alle Maßnahmen, die geeignet sind, die geforderte Qualität eines Softwareprodukts zu erreichen oder zu erhalten. Dies geschieht insbesondere hinsichtlich gewisser Qualitätskriterien [23], diese wären zum Beispiel: Korrektheit, Zuverlässigkeit, Wartbarkeit, Verständlichkeit, Verwendbarkeit und Dokumentation. Diese Kriterien sind maßgeblich für die Art der Qualitätssicherung und den erwähnten Maßnahmen. Daher umfassen diese Maßnahmen zum Beispiel die Aufstellung eines Qualitätsplans, die Planung von Reviews, Inspektionen und Tests und die Durchführung dieser Methoden. Des Weiteren wird auch die Festlegung auf Richtlinien und Standards sowie die begleitende Dokumentation darunter verstanden. Dieser kurze Überblick soll belegen, dass Qualitätssicherung ein wesentlicher Teil des Softwareproduktes ist.

3. Agile Softwareentwicklung

Die Agile Softwareentwicklung [1] ist eine relativ neue Methode innerhalb der Familie der Softwareentwicklungsprozesse. Diese Bezeichnung umfasst eine Fülle von Methoden, die sich durch eines klar gegenüber den herkömmlichen Methoden unterscheiden soll – Agilität. Mit dem Wort „agil“ soll die besondere Mobilität gegenüber gesteigerten Kundenanforderungen zum Ausdruck gebracht werden.

Agile Softwareentwicklungsprozesse oder „lightweight“ Prozesse, im Vergleich dazu spricht man von „heavyweight“ Prozessen im Sinne der traditionellen Softwareentwicklung, wurden mit dem Aufkommen von eXtreme Programming (XP) [6] [7] begründet. Durch den steigenden Anteil der Internetsoftware [8] wurde erkannt, dass herkömmliche Methoden mit ihren ausgeprägten Analyse- und Designphasen schwer für den schnellen Produktzyklus der im Internet vorherrscht geeignet sind. Außerdem können herkömmliche Methoden schlecht beziehungsweise gar nicht auf geänderte Kundenanforderungen während des Projektverlaufes reagieren. Dies ist hinsichtlich der Qualität eines Produktes aber wesentlich, da das Qualitätskriterium der Korrektheit insbesondere vom Verständnis des Kunden abhängig ist.

Agile Softwareentwicklung wird gerade in dieser Hinsicht als Allheilmittel, insbesondere in der Gemeinschaft der Verfechter der agilen Methoden, gesehen. Daher ist es schwer objektiv zu beurteilen ob sich durch den Einsatz von agilen Methoden hinsichtlich der Qualität Verbesserungen erzielen lassen können.

3.1 Basiskonzept und Manifest der agilen SE

Die wesentliche Strategie, die bei der agilen Softwareentwicklung betrieben wird, ist das Prinzip der Einfachheit, womit auf Änderungen so einfach wie möglich reagiert werden kann. Generell wird ein evolutionär-inkrementelles Vorgehen angewandt, nicht unähnlich der Entwicklung eines Prototyps. Beim traditionellen Ansatz liegt der Fokus auf der Dokumentation, bei der agilen Softwareentwicklung hingegen tritt an diese Stelle die Kommunikation. Umfangreiche Dokumentation ist weitaus weniger wichtiger, als das Gespräch zwischen allen Beteiligten. Diese Ansicht wird innerhalb der Verfechter der agilen Methoden soweit betrieben, dass der Quellcode einer Software als Dokumentation angesehen wird. Diese Fokussierung auf Kommunikation bedeutet, dass Kunde, Entwickler und andere Projektbeteiligte im ständigen Kontakt zueinander stehen. Dies bedeutet, dass alle Beteiligten einen aktuellen und hohen Wissensstand über das Projekt besitzen. Des Weiteren tritt anstatt der Orientierung an den Prozess eine Orientierung an das Team, so dass die Planung in den Hintergrund und die Zielorientierung in den Vordergrund gestellt wird [8]. Um dieses Basiskonzept auch niederzuschreiben wurde ein „Manifest der agilen Softwareentwicklung“ geschaffen [9] [10]. Diesem Manifest verpflichten sich alle Entwickler, die agile Methoden einsetzen. Das Manifest ist eine Sammlung von Grundsätzen die die Grundidee der agilen Softwareentwicklung repräsentieren. Diese Grundsätze lauten wie folgt:

- **Individuen und Interaktion** gegenüber von Prozessen und Werkzeugen
- **lauffähige Software** gegenüber von umfassender Dokumentation
- **Zusammenarbeit mit dem Kunden** gegenüber Vertragsverhandlungen
- **Reaktionen auf Veränderungen** gegenüber Befolgung eines Planes

Das Manifest der agilen Softwareentwicklung spiegelt die zentralen Werte der „agilen Bewegung“ [9] wieder. Aus diesem Werten kann die Abkehr von dem prozessorientierten Vorgehen hin zu einem den Entwickler und Programmcode in die Mitte stellenden Vorgehen erkannt werden. Daher kann die agile Softwareentwicklung als programmcodeorientierte Methode gesehen werden. Diese Verschreibung an ein Manifest lässt die agile Softwareentwicklung in den Schein einer Glaubensfrage rücken. Dieser Ansatz zeichnet sich in mancher Literatur nicht unähnlich der Streitigkeiten zwischen Linux- und Windowsverfechtern ab. Daher ist bei der agilen Softwareentwicklung meist davon die Rede, dass es sich um keinen neuen Prozess sondern eher um eine Einstellung handelt. Deshalb ist hier zu erwähnen, dass stets die nötige Objektivität bei den postulierten Vor- und Nachteilen zu beachten ist.

Durch die Anpassung bestehender Prozesse kann aus diesen ein agiler Prozess entstehen, diese Anpassung soll durch den situationsgerechten Einsatz von agilitätssteigernden Praktiken geschehen. Zu Beachten ist, dass nicht jeder Prozess dadurch zwangsläufig ein agiler Prozess wird. Ein agiler Prozess muss folgend Charakteristika aufweisen [11].

- Modularität des Entwicklungsprozesses
- Iteration über den Entwicklungsprozess, ermöglicht schnelle Fehlererkennung und -behebung
- Zeitliche Beschränkung der Iteration von einer bis zu sechs Wochen
- Sparsamkeit im Entwicklungsprozess entfernt alle unnötigen Aktivitäten
- Anpassungsfähig
- Inkrementeller Prozess
- Nicht auf den Prozess sondern auf die Beteiligten orientiert
- Teamarbeit

Zusammenfassend kann gesagt werden ein agiler Prozess ist inkrementell, kooperativ (Kunden und Entwickler arbeiten über die gesamte Projektdauer zusammen), einfach (schnell anzuwenden, leicht zu erlernen) und adaptierbar [8].

3.2 Agile Methoden

Im Folgenden werden ein paar agile Methoden kurz vorgestellt und auf deren Qualitätssicherungsmaßnahmen eingegangen. Extreme Programming wird eigens in einem Kapitel näher behandelt.

3.2.1 Scrum

Scrum wurde das erste Mal in einen Artikel von Takeuchi und Nonaka 1986 [14] erwähnt. Dort wurde es als adaptiver, schneller und selbstorganisierender Produktprozess vorgestellt. Der Begriff Scrum kommt aus dem Rugby und bezeichnet die Strategie einen Ball wieder ins Spiel zu bringen. Schwaber und Beedle [15] begründeten Scrum als agilen Softwareentwicklungsprozess. Scrum hat drei Phasen: Vorspiel, Entwicklung und Nachspiel, wobei die Vorspielphase in zwei Phasen unterteilt ist, in Planung und in Architekturdesign. Die Entwicklungsphase oder auch Spielphase genannt, ist der agile Teil von Scrum. In dieser Phase wird die Software durch so genannte Sprints entwickelt. Diese Sprints sind iterative Zyklen mit den traditionellen Phasen Bedarfsanalyse, Design, Entwicklung und Auslieferung. Innerhalb dieser Sprints können beliebige Techniken wie zum Beispiel eXtreme Programming zur Softwareentwicklung verwendet werden. In der Nachspielphase wird das Produkt fertig gestellt und ausgeliefert.

In Scrum hängen die zu setzenden Qualitätssicherungsmaßnahmen sehr stark von den in der Entwicklungsphase eingesetzten Techniken ab. Scrum ist geeignet für kleine Teams. Schwaber und Beedle [15] schlagen ein Team von fünf bis neun Mitglieder vor. Darüber hinaus sollten mehrere Scrumteams gegründet werden. Wie bei allen agilen Methoden fällt auch bei Scrum dem Team und den Teamrollen eine besondere Bedeutung zu. Das Scrumteam selbst hat die Verantwortung und die Autorität selbst Entscheidungen zu treffen und sich dementsprechend zu organisieren. Das Scrumteam ist auch für die Qualitätssicherung der einzelnen Sprints verantwortlich. Der Scrum Master ist für die Einhaltung der Spielregeln zuständig und dient als Kommunikationsschnittstelle zwischen Team, Management und Kunde. Der Kunde ist in die Entscheidungen für die Anforderung der Sprints eingebunden.

Scrum wird oft als Rahmengerüst für den Einsatz von XP gesehen, beziehungsweise als Projektmanagementumgebung für XP.

3.2.2 Feature Driven Development

Feature Driven Development wurde von John de Luca und Peter Coad [18] begründet. Dieser agile Ansatz deckt nicht den gesamten Softwareentwicklungsprozess ab, sondern konzentriert sich auf die Design- und Entwicklungsphase. Feature Driven Development wurde aber so ausgelegt, dass es mit andern Softwareentwicklungsprozessen zusammenarbeiten kann. Der Prozess ist so einfach wie möglich gehalten. Ausgehend von einem Modell wird eine Featureliste erstellt. Anhand dieser Liste werden diese Features geplant und anschließend in einem iterativen Prozess entwickelt. Die Summe der Features führt schließlich zu dem gesamtem Produkt. In den Iterationsphasen gibt es neben dem Unittesting als Qualitätssicherungsmaßnahme auch eine Code Inspektion. Die Rollenverteilung des Feature Driven Development weißt eine Testerrolle aus. Diese kann von einem Einzelnen oder von einem Team eingenommen werden Des Weiteren gibt es eine Rolle für die Dokumentation.

4. Traditionelle SE – Unterschiede zur agilen SE

Die Idee der traditionellen Softwareentwicklung ist im Wesentlichen an der Vorgehensweise anderer Ingenieurswissenschaften wie zum Beispiel dem Bauwesen [2] angelehnt. Der Schaffensphase geht eine Planungsphase voran, das heißt es wird zuerst anhand von Modellen das zu fertig stellende Produkt geplant und anhand dieser Modelle wird an die Konstruktion des Produktes herangegangen. Umgelegt auf den Softwareprozess bedeutet dies, dass der Softwareprozess aus Phasen besteht, in denen bestimmte Methoden verwendet werden um das Ziel – funktionierende Software – zu erreichen. Ein Phasenablauf könnte wie folgt aussehen: am Anfang steht eine Kundenanforderung, gefolgt von einer Planungs- und Designphase, nach der Annahme durch den Kunden folgt schließlich die Entwicklungsphase.

4.1 Traditionelle Softwareentwicklung

Wie bereits erwähnt spielt Planung beim traditionellen Ansatz eine große Rolle. Die Planung beschreibt also das Vorgehen innerhalb eines Softwareprojektes mit gewissen Randbedingungen. Wesentliche Bedeutung kommt hierbei auf die Wahl des Vorgehensmodells mit dem die Art der Softwareentwicklung bestimmt wird zu. Dieses Vorgehensmodell strukturiert die Softwareentwicklung in Phasen, wobei bei diesem Ansatz – getreu den Anleihen aus den Ingenieurswissenschaften – jede Phase zuerst abgeschlossen werden muss, bevor die nächste Phase be-

gonnen werden kann. Durch dieses Vorgehen soll gewährleistet werden, dass der Prozess überschaubar, planbar und vergleichbar wird. Dieses Vorgehen bedingt aber auch, dass keine Änderungen im Verlauf des Prozesses mehr auftreten. In der Realität können solche Änderungen nicht ausgeschlossen werden – daher ist auch hier ein großer Nachteil des traditionellen Ansatzes zu finden. Methoden die diesem Ansatz zu zuordnen wären sind zum Beispiel das Wasserfallmodell [3], das V-Modell [4] oder auch das Spiralmodell [5]. Des Weiteren werden traditionelle Ansätze dadurch gekennzeichnet, dass als Endergebnis jeder Phase ein Dokument entsteht. Diese Dokumentation dient zur Überprüfung des Erfolges, hinsichtlich der Planung, und fließt in das Produkt zum Beispiel in Form von Handbüchern ein. Ein weiteres Kriterium der traditionellen Ansätze ist, zumindest in den meisten Modellen, dass bei Erkennen eines Fehlers nur durch vermehrten Aufwand dieser Behoben werden kann beziehungsweise ein Rückschritt in eine frühre Phase auf Grund des Modells nicht gestattet ist.

4.2 Probleme und Qualitätssicherung beim traditionellen Ansatz

Wie Besprochen zeichnet sich der traditionelle Ansatz durch seine einzelnen Phasen aus. Qualitätssicherung ist hingegen ein Prozess der über die gesamte Projektdauer betrieben wird. Diese Eigenschaft macht es schwer Qualitätssicherung als eine einzelne Phase in den Prozess zu integrieren. Durch die Einführung von Phasen mit einer abschließenden Verifikations- und Validationsphase wurde zumindest der Qualitätssicherung insofern Rechnung getragen, dass erst durch den erfolgreichen Abschluss dieser Phase die nächste begonnen werden kann. Diese abschließende Phase ist in den einzelnen Methoden unterschiedlich realisiert, eines aber haben alle Methoden gemein – die Dokumentation. Die Dokumentation ist, bedingt durch die Planung und Analyse, ein zentrales Qualitätssicherungsmittel des traditionellen Ansatzes. In allen Modellen findet sich auch Testen als Qualitätssicherungsmaßnahmen wieder, meist als eigene Phase innerhalb des Prozesses. Testen wird als Mittel zur Testerkennung eingesetzt und kommt meist nach der Implementierungsphase zur Anwendung.

Dokumentation hingegen ist das Mittel, das eingesetzt wird um Fehler zu vermeiden. Dies bedingt natürlich eine genaue und umfangreiche Dokumentation. Voraussetzung ist, dass sich die Entwickler an diese Dokumentation halten und über diese Dokumentation untereinander kommunizieren. Gerade diese Vorgehensweise ermöglicht es den Prozess zu standardisieren und trägt hiermit wiederum einen Teil zur Qualitätssicherung bei. Dokumentation erhält diese wichtige Aufgabe dadurch, dass durch die Dokumentation alle Anforderungen, alle Schritte festgehalten werden und das an hand der Dokumentation der Reviewingprozess betrieben wird.

Hinsichtlich der Probleme ist gerade durch die Orientierung nach Phasen und nach dem Prozess zu bemerken, dass je früher ein Fehler erkannt wird, desto billiger ist es diesen zu beheben. Werden nun die geänderten Kundenanforderungen als Fehler betrachtet, Fehler hinsichtlich der Analysephase, so sind Änderungen je später sie bekannt werden umso teurer. Software ist in den letzten Jahren wie bereits erwähnt durch vermehrte Internetprojekte, aber auch durch die Dynamik des Marktes schnelllebiger geworden. Daher ist es schwieriger alle Anforderungen in der Analysephase zu erheben. Dies bedeutet eine genauere Anforderungsanalyse, vermehrter Aufwand bei der Dokumentation sowie den verstärkten Einsatz von neuen Softwaretechniken, die es erlauben Änderungen leicht einzubauen.

4.3 Agile vs. traditionelle SE

Vergleicht man nun den agilen Ansatz mit dem traditionellen Ansatz so scheint der agile Ansatz die Probleme des traditionellen Ansatzes zu lösen und auf die Änderungen hinsichtlich der Kundenanforderung zu reagieren. Ebenso ist durch den inkrementellen Entwicklungsprozess gewährleistet, dass Qualitätssicherung besser in den Gesamtprozess integriert werden kann. Andere Unterschiede ergeben sich aber auch hinsichtlicht den beteiligten Personen [12]. Der einzelne Entwickler hat durch den agilen Ansatz eine gute Kenntnis über das gesamte Projekt, im traditionellen Ansatz hingegen nur über seine zu lösende Aufgabe. Die Kommunikation trägt im agilen Ansatz dazu bei, dass Entwickler unterschiedlicher Module sich miteinander austauschen. In der traditionellen Softwareentwicklung geschieht dies nur durch die definierten Schnittstellen und über die Dokumentation. Dem Kunden kommt in der agilen Softwareentwicklung eine größere Bedeutung zu, er ist sozusagen vom Gedanken bis zur Geburt in das Projekt eingebunden und teilt seine geänderten Anforderungen stets den Entwicklern beziehungsweise dem Projektteam mit. Dadurch ergibt sich aber auch der Rahmen in dem agile Softwareentwicklung betrieben werden kann [12] [13]. Die Kommunikation unter den Entwicklern erfordert es, dass die Teamgröße beschränkt ist, zu große Teams erzeugen einen Overhead an Kommunikation. Dies schlägt sich auf die Projektgröße nieder, agile Softwareentwicklung kann nur bis

zu einer gewissen Projektgröße betrieben werden. Diese Einschränkung ergibt sich auch durch die angewandten Praktiken wie zum Beispiel Refactoring oder durch den inkrementellen Entwicklungsansatz. Agile Softwareentwicklung erzeugt Software, die kaum oder schwer wieder verwendbar ist – durch die starke Bindung an die Kundeanforderung und der kaum vorhandenen Planungsphase werden Einzellösungen produziert. Zudem wird angenommen, dass eine Neuentwicklung wesentlich günstiger ist als eine Weiterentwicklung.

Nachstehende Tabelle soll einige Unterschiede nochmals verdeutlichen.

	agile SE	traditionelle SE
Beispiele	Scrum, FDD, XP	V-Modell, Spiralmodell
Ziel	minimale Kosten bei maximaler Kundenzufriedenheit und maximaler Qualität	
Strategie	Änderung so einfach wie möglich	Fehler beziehungsweise Änderungen vermeiden
Mittel	evolutionäres inkrementelles Vorgehen	Dokumentation und Prüfung
Prozess	gestaltbar, teamorientiert	zertifizierbar, planbar, wiederholbar

Tabelle 1: Unterschiede agile und traditionelle SE

5. eXtreme Programming

In den vorangegangen Kapitel wurde eXtreme Programming bereits als Auslöser für die Entstehung der agilen Softwareentwicklung genannt. Extreme Programming (XP) wurde von Kent Beck [19], Ward Cunningham und Ron Jeffries begründet. XP wurde als Antwort zu den bestehenden Problemen des traditionellen Ansatzes entwickelt und setzte sich als der bekannteste Vertreter der agilen Methoden durch. Im Wesentlichen ist XP eine Sammlung von Ideen und Praktiken wobei durch die extreme Anwendung dieser Methoden der Technik der Name verliehen wurde.

5.1 Der Lifecycle von XP

Der Prozess von XP besteht aus fünf Phasen: Exploration, Planning, Iterations to Release, Productionizing, Maintenance und Death.

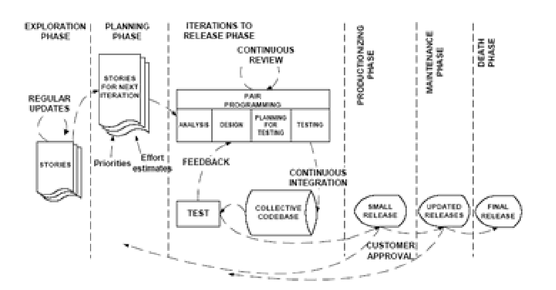


Abbildung 1. Lifecycle XP-Prozess [8]

Im Folgenden werden die einzelnen Phasen näher beschrieben [8].

In der „Exploration“ Phase werden die gewünschten Anforderungen anhand so genannter Storycards erfasst. Diese Storycards werden vom Kunden ausgefüllt und beinhalten die Funktionalität der gewünschten Software. Das Projektteam macht sich zwischenzeitlich mit den verwendeten Tools und Softwarepraktiken vertraut und baut damit einen ersten Prototypen des zu entwickelnden Systems. Die Explorationsphase dauert in der Regel zwischen ein paar Wochen und ein paar Monaten, abhängig von der Projektgröße und der Vertrautheit mit den eingesetzten Tools. Die „Planning“ Phase dauert ein paar Tage und beinhaltet die Planung der einzelnen Storycards. Des Weiteren wird in dieser Phase eine Vereinbarung mit dem Kunden getroffen welche Funktionalitäten die erste Version der zu entwickelnden Software zur Verfügung stellen soll. In der „Interations to release“ Phase wird nun diese erste Version entwickelt. Diese Phase beinhaltet mehrer Zyklen die in der Regel bis zu vier Wochen dauern können. Die erste Iteration liefert die Architektur des Systems. Danach wird vom Kunden ausgewählt welche Storycard bei der nächsten Iteration implementiert wird, diese Version wird anhand von Kundentestfällen geprüft. Nach der letzten Iteration ist das Produkt fertig für die nächste Phase. In der „Productionizing“ Phase wird die Software weiteren Tests unterzogen. Innerhalb dieser Phase können neue Kundenanforderungen identifiziert werden. Hier muss die Entscheidung getroffen werden, ob diese Anforderungen in die bestehende Version eingebaut werden, dies bedeutet ein erneutes durchlaufen von Iterationen oder ob diese Anforderungen erst während der nächsten Phase implementiert werden, diese Anforderungen müssen auf jeden Fall dokumentiert werden. Nachdem das Produkt diese Phase bewältigt hat wird das Produkt an den Kunden ausgeliefert und die nächste Phase „Maintenance“ beginnt. Die „Maintenance“ Phase zeichnet sich dadurch aus, dass die Software bereits beim Kunden vor Ort im Einsatz ist, für etwaige Änderungen werden wieder Iterationen

laut der „Iterations to release“ Phase durchgeführt, wobei hier die Entwicklungszeiten kürzer werden. Die „Death“ Phase wird im XP-Lifecycle dann erreicht, wenn durch den Kunden keine weiteren Anforderungen gestellt werden oder das System auf Grund von Fehlentwicklungen nicht mehr einsetzbar ist. In diese Phase fallen auch die Dokumentation und die Übergabe an den Kunden.

5.2 Das XP-Team

Wie bei allen agilen Methoden kommt auch bei XP dem Team eine besondere Bedeutung [19] zu. Wobei dem Kunden innerhalb des Prozesses eine besondere Rolle zukommt. Der Kunde ist sowohl für das Verfassen von den Anforderungen als auch für die Erstellung von Testfällen zuständig. Zu diesem Zweck wird ihm innerhalb des Prozesses ein Tester zur Seite gestellt, damit die Testfälle aufbereitet werden können. Diese Testfälle werden sowohl vom Kunden, Tester und Programmierer ausgeführt. Der Kunden hat des Weiteren die Aufgabe den nächsten Implementierungsschritt zu bestimmen, das heißt wie bereits erwähnt entscheidet der Kunde welche Anforderung im jeweiligen Iterationszyklus erfüllt werden muss. Für die Qualitätssicherung bedeutet dies, dass Testfälle beziehungsweise das Testen im Gegensatz zu herkömmlichen Methoden eine zentrale Rolle einnimmt. Dies besonders dadurch, dass die Kundeneinbindung ein Teil der Qualitätssicherung geworden ist. Weitere nennenswerte Rollen innerhalb des Teams wären der Tracker, der Coach und nicht zu vergessen der Programmierer. Der Tracker überwacht den Verlauf des Projektes und überprüft ob das Ziel mit den vorhandenen Ressourcen erreichbar ist. Der Coach ist für den gesamten XP-Prozess verantwortlich und versucht die Teammitglieder bei der Befolgung des Prozesses zu unterstützen. Der Programmierer ist für die ihm zugeteilten Programmiertätigkeiten und für die Erstellung und Ausführung geeigneter Testfälle zuständig.

5.3 XP-Praktiken

Das wesentliche an XP ist aber nicht die Art des Prozesses sondern die eingesetzten Techniken. Im Folgenden werden XP-Praktiken vorgestellt.

Planungsspiel: Das Planungsspiel ist der Ersatz für das Pflichtenheft. Im Planungsspiel wird der Funktionsumfang der Software festgelegt und die Aufwandbetrachtungen von den Entwicklern werden erfragt. Die User Stories werden wie bereits erwähnt vom Kunden auf Karten aufgeschrieben und nach Wichtigkeit klassifiziert. Die Entwickler schätzen den Aufwand.

Coding Standards: Die Entwickler definieren Coding Standards, an die sich allen im Verlauf des Projektes halten müssen. Im Unterschied zu anderen Softwareentwicklungsprozesse werden diese Standards von den Entwicklern selbst festgelegt. Diese Standards sollen zur Qualitätssicherung beitragen, außerdem soll es ermöglicht werden das Einarbeitungszeiten bei Ausfällen oder Wechseln kurz ausfallen.

Pair Programming: Bei jedem Entwicklungsschritt, bei XP wird von Tasks gesprochen, werden zwei Entwickler benötigt. Diese Anforderung soll auf der einen Seite sicherstellen, dass die Standards eingehalten werden und auf der anderen Seite soll dadurch zur Fehlervermeidung beigetragen werden. Getreu nach dem Motto „Vier Augen sehen mehr als zwei“. Des Weiteren wird sichergestellt, dass mehrere Programmierer Kenntnisse über den Code besitzen um so Ausfällen vorzubeugen. Die Programmierer wechseln sich untereinander ab, außerdem tauschen die Paare sich aus. Diese Praktik wird oft als Verschwendung von Ressourcen gesehen und gilt daher als umstrittenste Praktik von XP. Sie wird jedoch innerhalb von XP als wesentliches Mittel zur Qualitätssicherung verstanden.

Unit Tests: Die Tests werden vor Beginn eines Tasks geschrieben, dies soll dem Programmierer nochmals verdeutlichen welche Funktionalität bereitgestellt werden soll. Des Weiteren wird sichergestellt, dass die Implementierung der gewünschten Funktionalität gerecht wird.

Functional Tests: Der Kunde entwickelt Funktionstests (Akzeptanztest), dies geschieht je nach Kenntnis des Kunden. Entweder entwirft der Kunde diese Tests selbst oder mit Hilfe eines Testers beziehungsweise eines Entwicklers. Anhand dieser Tests werden die User Stories überprüft und vom Kunden als erfüllt angesehen.

Collective Code Ownership und Refactoring: Diese Praktik bezeichnet, dass der fertig gestellte Code nicht im Besitz des jeweiligen Entwicklerpaares sondern im Besitz des Teams ist. Das heißt, bei notwendiger Überarbeitung (Refactoring) von Programmteilen soll dies unter Einhaltung der definierten Standards sofort und unmittelbar geschehen und kann von jedem Entwickler durchgeführt werden. Voraussetzung hierbei ist, dass bereits bestandene Tests weiterhin zu 100% erfüllbar sind.

Continuous Integration: Fertiggestellte Tasks werden fortlaufend in das Gesamtsystem integriert. Integration steht also nicht wie bei dem traditionellen Ansatz am Ende der Entwicklung sondern ist ein fortschreitender Prozess. Natürlich müssen nach der Integration alle bereits definierten Tests wieder erfüllt werden. Diese fortlaufende Integration soll gewährleisten, dass dem Kunden jederzeit eine Gesamtsicht über das Produkt gegeben werden kann.

5.4 Qualitätssicherungsmaßnahmen bei XP

Wie anhand der Praktiken von XP ersichtlich ist Qualitätssicherung in XP hauptsächlich [20] durch die Verwendung von Unit- und Funktionstests realisiert. Des Weiteren ist die Praktik des Pair Programming ein wesentlicher Qualitätssicherungsfaktor. Durch den Einsatz von Pair Programming bekommt XP Elemente von Reviewing, da sich die Programmierpaare abwechseln. Ein jedoch nicht unwesentlicher Faktor ist der Kunde als Qualitätssicherungsinstrument. Durch die Einbindung des Kunden in den Qualitätssicherungsprozess wird gewährleistet, dass seine Anforderungen erfüllt und überprüft werden. Wie bereits erwähnt unterscheiden sich agile Methoden hinsichtlich der Dokumentationstätigkeit gegenüber den traditionellen Methoden. Es scheint, dass in XP kein Platz für Dokumentation besteht. Die Relevanz einer Dokumentation [21] wird aber bei genauerer Betrachtung bewusst, vor allem wenn Kommunikation in den Vordergrund tritt. Kommunikation bedingt ab einer bestimmten Projektgröße Dokumentation - besonders wenn diese durch Vergaberichtlinien oder Standardisierungsmaßnahmen erforderlich ist [20]. XP versucht dieses Fehlen durch die Codierungsstandards und den Kundenkontakt wettzumachen. Hierbei ist hängt es aber sehr stark von dem Kunden ab, ob dieser auf Dokumentation verzichten kann beziehungsweise auf wie viel Dokumentation der Kunde besteht. XP in der Grundidee verzichtet zu Gunsten von Testfällen auf ausführliche Dokumentation, dort wo es erwünscht ist ist die Verwendung bei Bedarf jederzeit möglich.

5.5 Vor- und Nachteile

XP verspricht auf Grund der Einfachheit die Entwicklung von leichtverständlichem Code, dies ist durch die Einhaltung von Programmierstandards und der starken Codezentriertheit möglich. Vorausgesetzt ist hierbei die strikte Einhaltung der Standards sowie ausreichende Codedokumentation anhand von Kommentaren. Teamarbeit soll ein breites Verständnis der einzelnen Teammitglieder gegenüber dem Produkt schaffen und so zu erhöhter Qualität und Prozessbeschleunigung führen. Der Vorteil der Teamarbeit kann sich aber durch die erhöhte Kommunikation besonders bei größeren Projekten leicht zu einem Nachteil verkehren. Der wesentlichste Vorteil hinsichtlich der Qualitätssicherung ist aber die ständige Durchführung von Tests. Die starke Einbindung des Kunden in den Verlauf der Softwareentwicklung soll die Erfüllung der Anforderungen gewährleisten. Hierbei ist aber zu beachten, dass Kunde nicht gleich Auftraggeber sein muss. Der Kunde ist im Falle der Softwareentwicklung neben dem Auftraggeber, der den Bedarf der Software erkennt und das Projekt initialisiert, der Endanwender. Eine genaue Identifikation und Einbindung entsprechender Keyuser ist maßgeblich für den Erfolg von XP Projekten beziehungsweise bei agilen Projekten. Besonders weil XP davon ausgeht, dass der Kunde weiß was er will und eine genaue Vorstellung der Software hat. Eine Anforderungsanalysephase existiert in keiner herkömmlichen Form und anhand der Storycards muss der Kunde nach und nach sich ein Bild der Software zurechtlegen. XP vermeidet jede zeitintensive Tätigkeit hier besonders die Dokumentation. Dies kann insbesondere Probleme hinsichtlich der Auftragvergabe bereiten. Die Dokumentation beziehungsweise das Handbuch wird bei XP erst bei Beendigung des Projektes den Kunden übergeben, dies mag vielleicht viele Kunden davon abhalten sich auf Projekte mit XP einzulassen, da die Software zuvor schon lange in Verwendung ist beziehungsweise getestet wird.

6. Qualitätssicherung bei agiler Softwareentwicklung

In den vorangegangen Kapitel wurde bereits sehr viel über Qualitätssicherung gesprochen. Dieses Kapitel soll die wesentlichen Punkte der Qualitätssicherung bei agiler Softwareentwicklung behandeln und auf die Testarten bei der Anwendung von agilen Methoden eingehen.

Durch die Umorientierung des Prozesses von Dokumentation und definierten Tests- und Reviewingphasen, hin zu einem inkrementellen Ansatz erfolgt hinsichtlich der allgemeinen Qualitätssicherung eine Stärkung der Tests. In der agilen Softwareentwicklung werden Dokumente als den Prozess behindernd angesehen, dies bedeutet aber nicht, dass keine Dokumentation stattfindet. Rahmendokumente sowie zum Beispiel das Handbuch werden weiterhin erstellt beziehungsweise müssen weiterhin erstellt werden. Die Abkehr von der Dokumentation hin zu der wesentlichen Arbeit des Programmierers lässt auch

erahnen dass agile Softwareentwicklung von diesen auch betrieben und forciert wird. Im Zusammenhang der agilen Softwareentwicklung insbesondere in der Literatur wird gerade die Abkehr von der Dokumentation hin zur Kommunikation und Tests als der Meilenstein der agilen Entwicklung beschrieben. Die Hauptaufgabe der Qualitätssicherung ist daher nun das Testen. Durch den Wegfall klassischer Analysephasen ist nun der Kunde dazu berufen die Rolle des Anforderungsanalysten sowie die Rolle des Testers teilweise auszuüben. Der Kunde sowie der Entwickler erstellen nun Testfälle, die das Produkt am Ende der einzelnen Iterationsphasen erfüllen muss. In diesem Zusammenhang spricht man hier von einer „Testkrankheit der Entwickler", da nun das Testen nicht mehr nur für die Tester zum Aufgabengebiet zählt sondern auch vermehrt zu dem des Entwicklers. Die Art der Qualitätssicherung beziehungsweise des Testens ist aber von der verwendeten agilen Methode abhängig. Im Wesentlichen kann aber davon gesprochen werden, dass die Testtätigkeit beim Einsatz von agilen Methoden nicht mehr als notwendiges Übel angesehen wird, sondern dass von einer testgetriebenen Softwareentwicklung gesprochen werden kann. Diese fortwährende Testgenerierung und –ausführung soll nun dazu dienen die Kundenanforderungen zu überprüfen und zu erfüllen. Diese ständige Tätigkeit kann nur ausgeübt werden wenn entsprechende Automatisierung vorgenommen werden kann. Die Voraussetzung der Automatisierbarkeit der Tests ist ein wesentliches Kriterium für die Entwicklung mit agilen Methoden. Der Einsatz von automatisierten Testframeworks soll neben der leichten Überprüfung der Testfälle auch dazu dienen den Kunden ein Werkzeug zu geben, um die von ihm gestellten Anforderungen in Testfälle umzusetzen. Um diese Anforderungen zu erfüllen können vier Testtypen für die agile Softwareentwicklung als relevant betrachtet werden [22], welche im Folgenden betrachtet werden sollen.

6.1 Akzeptanztest

Das Ziel der Akzeptanztest ist es die Erfüllung der Kundenanforderung zu prüfen. Akzeptanztests werden so wie das später erwähnte „By-Hand-Testing" vom Kunden ausgeführt. Wesentlich bei Akzeptanztests ist, dass der Kunde keine unmittelbare Kenntnis über die eingesetzte Softwaretechnik besitzen muss. Anhand der von ihm erwartenden Funktionalität spezifiziert der Kunde die Testfälle und führt diese aus. Die Durchführung der Tests hängt dabei von dem Wissenstands des Kunden ab, deshalb steht den meisten Kunden bei der Spezifizierung und Durchführung der Tests ein Tester zur Seite. Akzeptanztests werden fortlaufend durchgeführt, das heißt je nach eingesetzter Methode wird innerhalb der Entwicklungsphase fortlaufend mittels Akzeptanztest geprüft ob die Anforderungen des jeweiligen Iterationsschrittes erfüllt werden. Da der Kunde im Besitz des Akzeptanztests ist obliegt es alleine dem Kunden zu bewerten ob der Akzeptanztests erfolgreich absolviert wurde. Bei Scheitern des Testes wird dem Kunden das Testprotokoll zur Kontrolle vorgelegt. Die Erfüllung des Akzeptanztests ist wesentlich für die Entscheidung ob der nächste Prozessschritt ausgeführt werden kann oder nicht.

6.2 By-Hand Testing

So genannte „By-Hand"-Tests werden dazu verwendet um die Funktionalität von graphischen Eingabemasken zu testen. Diese Art von Tests soll vor allem mögliche Eingabefolgen von Kunden hingehend auf Richtigkeit und Stabilität überprüfen. Hierzu werden typische Eingabefolgen von Kunden ausgeführt, diese Eingabefolgen werden aufgezeichnet und dann mittels Skript durchgeführt. Da diese Art von Tests sehr zeitaufwendig ist, wird sie nur auf Eingaben angewandt die im späteren Einsatzumfeld der Software häufig Anwendung finden. Außerdem können diese Tests erst relativ spät durchgeführt werden, da die wesentlichen darunter liegenden Funktionalitäten vorhanden sein müssen. „By-Hand"-Tests werden vor allem bei klassischen Internetprojekten verwendet, da hier die Eingaben und die dementsprechenden Funktionen erfolgskritischer als bei anderen Projekten zu betrachten sind.

6.3 Unit Tests

Im Gegensatz zu den vorangegangenen Testarten werden Unit Tests von den jeweiligen Entwicklern durchgeführt. Wie im Kapitel über XP bereits erwähnt werden Unit Tests vor Beginn der Entwicklung definiert um so das erwartende Verhalten der Software im Vorfeld festzulegen. Um einen Unit Test erfolgreich abzuschließen müssen alle Erfordernisse erfüllt werden, das heißt im Gegensatz zum Akzeptanztest ist der Erfolg von der definierten Funktionalität abhängig und nicht von der ausführenden Person. Da in der agilen Softwareentwicklung ein evolutionärer Ansatz vorherrscht bedeutet dies, dass ein Unit Test zu jedem Zeitpunkt in der Entwicklung zu 100% erfüllt werden muss. Ist dies nicht der Fall so muss dies sichergestellt werden bevor die Entwicklung vorangetrieben werden kann – in diesem Zusammenhang spricht man von

Regressionstests. Diese Gesamtsicht macht die Ausführung von solchen Tests kompliziert, da stets die Testfälle für das gesamte Produkt konzipiert werden müssen. Um dies zu erleichtern kann sich der Entwickler den bereits erwähnten Testframeworks bedienen. Frameworks generieren automatische Testfälle und überprüfen die Ausführung. Der Einsatz solcher Werkzeuge ist aber mit Vorsicht zu genießen, da getreu der Forderung – Testfälle vor dem Entwickeln zu definieren – eine Verwendung von solchen Frameworks kaum möglich ist. Trotz dieser Einschränkung können solche Frameworks, vor allem bei Tests über das gesamte Produkt, sehr hilfreich sein. Man kann hier von einer 80-20 Regel ausgehen, 20% automatisierte Tests und 80% vom Entwickler erstellte Tests. Unit Tests, obwohl näher bei der Software, sind gleich bedeutend wie Akzeptanztest und sollten daher nicht höher bewertet werden, um so der Gefahr der Vernachlässigung der Akzeptanztest zu entgehen.

6.4 Performance Tests

Performance Tests dienen dazu die Leistung des Softwaresystems zu quantifizieren. Diese Art von Tests wird von den Entwicklern oder Testern durchgeführt um die Anforderungen hinsichtlich der Leistung zu überprüfen und zu bewerten. Hinsichtlich der Qualitätssicherung sollen die Performance Tests das Kriterium der Effizienz überwachen.

6.5 Die Rolle des Testers

Im Gegensatz zum traditionellen Ansatz, wo die Rolle des Testers anhand der Prozessstruktur klar definiert ist, scheint sie bei Verwendung von agilen Methoden sehr beschnitten beziehungsweise kaum vorhanden. Da die Spezifizierung und Durchführung von Tests vermehrt vom Kunden und dem Entwickler durchgeführt wird, stellt sich die Frage - Haben Tester in der agilen Softwareentwicklung noch Platz?

Der Tester im herkömmlichen Sinn, das heißt Modellierung der Testfälle anhand der Anforderung an das Produkt, erstellen eines Testberichtes, erneutes Testen usw. scheint überflüssig geworden zu sein. Die Rolle des Testers in einem agilen Softwareentwicklungsprozess ist vielmehr eine zwiespältige, nämlich die eines Kunden und Entwicklers. Der Tester ist durch die Art des Prozesses in die Entwicklung involviert, ist aber zugleich dazu angehalten die Sicht des Kunden nicht zu verlieren. Zu diesem Zweck ist der Tester meist bei der Erstellung der Akzeptanztest dem Kunden beigestellt, dies geschieht in erster Linie um den Kunden bei der Bedienung von Testtools und bei der Formulierung der Testfälle behilflich zu sein. Im weiteren Verlauf ist der Tester für die Auswertung und Aufbereitung der definierten Tests zuständig. Bei den unterschiedlichen Methoden der agilen Softwareentwicklung ist es durchaus möglich, dass Tester die fast originäre Form ihrer Tätigkeit ausüben und in einer eigenen Testphase Testfälle generieren und prüfen. Bei XP hingegen ist die Rolle des Testers durch den Prozess neu zu definieren und eher als Qualitätssicherungsverantwortlicher zu sehen.

7. Fazit

Agile Softwareentwicklung beschreitet im Gegensatz zum traditionellen Ansatz einen vollkommen anderen Weg wie Software entwickelt werden kann. Diese Art von Entwicklung macht zwar keine vollkommen neue Art von Qualitätssicherung notwendig, aber beeinflusst die Art wie Qualitätssicherung betrieben wird. Im agilen Ansatz wird die Qualitätssicherung verstärkt als ein Instrument zur Überprüfung der Kundenanforderung gesehen und beeinflusst damit direkt die Qualität des Produktes. Die unterschiedliche Art der Qualitätssicherung liegt nur im Fokus der Qualitätssicherung, Tests werden genauso wie beim traditionellen Ansatz eingesetzt, ebenso Dokumentation und Reviewing. Die Entscheidung welche Art von Prozess verwendet werden soll ist von anderen Faktoren abhängig, nämlich von der Projektgröße und ob eine bestimmte Methoden betrieben werden muss. Agile Softwareentwicklung ist für die Entwicklung von kleinen und mittleren Projekten gut geeignet, besonders aber für die Entwicklung von Einzellösungen wie zum Beispiel von Internetprojekten. Gerade anhand dieses Beispieles wird offensichtlich, dass hier die übertriebene Plantreue eher dem Projekt hinderlich ist und eine Zuwendung hin zu einer leichtgewichtigen Form der Qualitätssicherung gefunden werden muss. Zusammenfassend kann gesagt werden, dass die agile Softwareentwicklung eine alternative zu den traditionellen Ansätzen sein kann. Die Qualität hingegen hängt von der Durchführung des Prozesses ab.

8. References

[1] Cockburn A., *Agile Software Development*, Addison-Wesely, Boston, 2001

[2] Fowler M., *The New Methodology*, http://www.martinfowler.com/articles/newMethodogy.html, Besucht am 12. April 2004

[3] Royce W.W., *Managing the Development of Large Software Systems*, Proceedings of IEEE WESCON, August 1970

[4] Boehm, B., Guidelines for verifying and validating software requirements and design specification. Euro IFIP 1979, pp. 711 - 719

[5] Boehm B., *A Spiral Model of Software Development and Enhancement*, ACM SIGSOFT Software Engineering Notes, August 1986

[6] Beck K., *Embracing Change with Extreme Programming*, IEEE Computer 32(10), pp. 70 - 77

[7] Beck K., *Extreme programming explainend: Embrace change*, Addison-Wesely, Boston, 1999

[8] Abrahamsson P., Salo O., Rankainen J., Warsta J., *Agile software development methods – Review and analysis*, VTT Electronics, 2002

[9] Agile Alliance, http://www.agilealliance.com, Besucht am 12. April 2004

[10] Agile Manifesto, http://agilemanifesto.org, Besucht am 12. April 2004

[11] Miller G. G., *The Characteristics of Agile Software Processes*, The 39th International Conference of Object-Oriented Languages and Systems (TOOLS 39), Santa Barbara, CA., July 29 - August 03, 2001

[12] Boehm B., *Get Ready For The Agile Methods, With Care,* Computer 35(1), pp. 64-69

[13] Turk D., France R., Rumpe B., *Limitations of Agile Software Process*, XP 2002, 2002

[14] Takeuchi H., Nonaka I., *The New Product Development Game*, Havard Business Review, Jan./Feb. 1986, pp. 137 - 146

[15] Schawber K., Beedle M., *Scrum Development Process*, OOPSLA'95 Workshop on Business Object Design and Implementation, Springer Verlag, 1995

[16] Cockburn A., *Writing Effective Use Cases. The Crystal Collection for Software Professionals*, Addison-Wesley Professional, Boston, 2000

[17] Cockburn A., *Surviving Object-Oriented Projects: A Manger's Guide*, Addison Wesley Longman, 1998

[18] Coad P., LeVebvre E., De Luca J., *Java Modeling In Color With UML: Enterprise Components and Process*, Prentice Hall, 1999

[19] Beck K., *Extreme Programming explainend: Embrace change*, Addison-Wesley, 1999

[20] Theunissen W.H., Derrick G., *Standards and Agile Software Development*, SAICSIT 2003, Johannesburg, September 2003

[21] Forward A., Lethbridge T.C., *The Relevance of Software Documentation, Tools and Technologies: A Survey,* ACM Symposium on Document Engineering 2002: pp. 26 - 33

[22] Marchesi M., Succi G., Wells D., Williams L., *Extreme Programming Perspectives*, Addison-Wesley, Boston, 2002

[23] Jalote P., An Integrated Approach to Software Engineering 2nd ed., Springer, 1997 p.11

Software im Automobil – Qualitätssicherung durch MISRA

Gauster Brigitte
0060077
bgauster@edu.uni-klu.ac.at

Abstract

In dieser Arbeit wird Ihnen anfangs ein Einblick über die Verwendung von Software in den Kraftfahrzeugen gegeben und welche Auswirkungen ein Versagen dieser Systeme haben kann. Durch die angeführten Fehlfunktionen soll die Bedeutsamkeit der Qualität der eingebauten Software hervorgehoben und des Weiteren spezielle Verfahren und Metriken zur Sicherung dieser vorgestellt werden. Das Hauptaugenmerk richtet sich hierbei auf MISRA – The Motor Industry Software Reliability Association, die besondere Richtlinien für den Automobilsektor entwickelt hat, um eine Hilfestellung bei der Entwicklung von „life-critical" Software zu geben.

Abschließend wird Ihnen noch ein kleiner Blick in die Zukunft gewährt und einige „neu anrollende Techniken" vorgestellt.

1. Einleitung

Das Automobil ist weltweit zu einem Symbol für Fortschritt und Entwicklung geworden und zugleich ein Statussymbol für die meisten Bürger. Undenkbar erscheint es für uns, kein Fahrzeug zu besitzen, um damit von einem Punkt zum anderen zu gelangen.

Doch heute ist der PKW nicht nur mehr ein Fahrzeug. Ausgestattet mit neuen mechanischen Bauteilen und software-gesteuerten Komponenten wurde das Fortbewegungsmittel zum Arbeitsplatz, zur Freizeitbeschäftigung und zum Hobby.

Das Fahrzeug von heute beinhaltet inzwischen mehr Software (SW) als man denken könnte. In den aktuellen Mittelklassewagen gehören Extras wie Klimaanlage oder Tempomat zur Grundausstattung, die durch eigens entwickelte Computerprogramme gesteuert werden und bei einem Neukauf eines Autos kaum noch wegzudenken sind. Navigationssysteme, Einparkhilfen, ABS-controlling und vieles mehr ist für den rasanten Fortschritt und die Beliebtheit moderner Fahrzeuge verantwortlich – vorausgesetzt diese funktionieren!

Um einen Überblick über die Entstehungsgeschichte der eingebetteten Softwaresysteme im Auto zu geben, soll die nachfolgende Abbildung dienen, in der einerseits ersichtlich ist, welche Elektronik wann ihren Ursprung hatte und andererseits, wie umfangreich der Einsatz dieser mit der Zeit wurde.

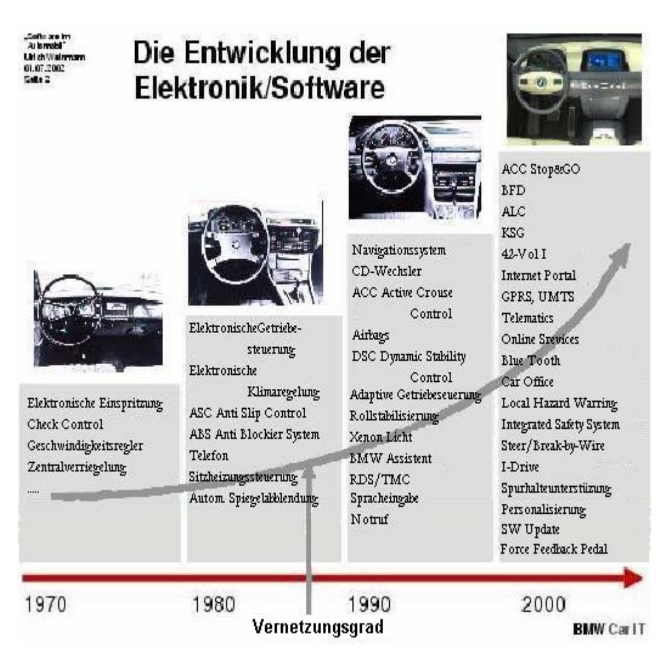


Abbildung 1: Die Entwicklung der Elektronik und Software im Automobil [10]

Des Öfteren wird der Begriff Elektronik fallen, wobei es wichtig ist, diesen vorerst zu definieren, damit der Zusammenhang zur verwendeten Software hergestellt werden kann.

„Elektronik: Im Mittelpunkt dieses Zweiges stehen u. a. der Entwurf und die Anwendung von elektronischen Schaltungen in Geräten – umgangssprachlich bezeichnet man diese Schaltungen ebenfalls als Elektronik. Elektronische Schaltungen realisieren verschiedene Funktionen zur Informationsverarbeitung, außerdem werden logische Operationen, wie z. B. die elektronischen Abläufe in

einem Computer, durch elektronische Schaltungen verwirklicht.“ [28]
Eine besondere Wechselbeziehung der Elektronik besteht zur Informatik; zum einen liefert die Elektronik die für die angewandte Informatik notwendigen technischen Grundlagen elektronischer Rechenmaschinen, andererseits ermöglichen die Verfahren der Informatik erst komplexe elektrotechnische Systeme.

2. Heutiger Standpunkt

„Experten nach wird der weltweite Markt der Automobilindustrie in den nächsten Jahren von 125 auf 270 Mrd. Euro steigen und so auch der Prozentsatz der eingebauten Software im Auto.
Der Software-Anteil soll im Jahr 2010 zirka 13 Prozent oder durchschnittlich 1.450 Euro des Fahrzeug-Gesamtwertes umfassen. Laut Prognose wird sich der Markt für Software im Vergleich zum Jahr 2000 auf rund 100 Millionen Euro vervierfachen.“ [9]
Bezeichnend dafür ist bereits die jetzige Anzahl der eingebauten elektronischen Steuergeräte in den neu angefertigten Mittelklassewagen – dieser Anteil beträgt inzwischen 20 bis 65 Stück verschiedener Software-Komponenten.

Durch diesen immer größer werdenden Anteil ist die Software zu einem wettbewerbsentscheidenden Faktor zur Steigerung des Marktimages geworden und erfordert intensive Zusammenarbeit von Forschungspartnern aus der Automobil- und Softwareindustrie.

Der Kunde will ein Massenprodukt, das billig und dennoch zuverlässig ist und eine lange Lebensdauer hat. Sofort eintreffende Verkehrsmeldungen und die Möglichkeit die Mautgebühren automatisch bezahlen zu können, sind dabei nur kleinere Wünsche.
Die Karosserie selbst unterscheidet sich hierbei bezüglich ihrer „Haltbarkeit“ gewaltig von der eingebetteten Software. So ist der Lebenszyklus eines KFZs und dessen mechanischen Bauteilen von einigen Services, gelegentlichen Erneuerungen von Verschleißteilen und eventuellen „optischen“ Verbesserungen geprägt.
Die Software hingegen durchläuft ganz andere Phasen. Allein die Beständigkeit ist eingeschränkter und die nötigen Updates und Upgrades sind meist unmöglich durchzuführen, da diese einen gesamten Ausbau aus dem Systemumfeld mit sich ziehen würden.

2.1. Einsatzbereiche der Software

Die wesentlichen Software-Komponenten, die in einem KFZ eingebaut sind, lassen sich allgemein in sicherheitskritische und nicht sicherheitskritische Systeme einteilen. Wobei das Versagen der ersten Einteilung Menschenleben fordern kann, sind nicht sicherheitskritische Systeme mehr für den Komfortbereich verantwortlich und strapazieren lediglich die Nerven des Autobesitzers.
Dadurch stellt sich die Frage welche Software im PKW unterstützt uns Autofahrer und wird als unabdingbar angesehen und welche „unterhält und amüsiert“ uns nur?

Eine der wohl wichtigsten Erfindungen der letzten Jahre im Zusammenspiel mit Software war die des Airbags (patentierte Erfindung der Firma Mercedes von 1968/69). Hier übernimmt die Software die Steuerung von eigens angefertigten Airbag-Chips. Diese Steuerung und Überwachung ist bei weitem wichtiger, als man annehmen könnte, denn die Aktivierung des Airbags darf nicht in allen Situationen erfolgen. Bei einem Frontalaufprall misst die Software die Aufprallgeschwindigkeit und den Aufprallwinkel und errechnet in 10 bis 40 tausendsel Sekunden ob es sich dabei um eine lebensbedrohliche Kollision handelt, oder der Lenker nur einen kleinen Parkschaden verursacht hat.

Auch das ABS (Anti-Blockier-System) ist heute eine Grundausstattung im Automobil, die nicht mehr wegzudenken ist. Sensoren an jedem Rad messen die Drehgeschwindigkeit und lösen gegebenenfalls die Bremse, um ein Blockieren zu verhindern, so den Bremsweg zu verkürzen und die Lenkbarkeit des Fahrzeuges noch möglich zu machen.

Das ESP (Elektronisches-Stabilitäts-Programm) ist eine weitere Entwicklung die schon so manches Menschenleben retten konnte. Daten aus Lenkwinkelsensoren am Lenkrad werden mit den Daten aus Raddrehzahlsensoren verglichen und so kann die SW erkennen, in welche Richtung der Fahrer sich fortbewegen will. Entspricht aber die tatsächliche Fahrtrichtung (sei es Aufgrund von Eis oder Schnee) nicht die der gewünschten, greift die Software in einem Bruchteil einer Sekunde in das Brems- und Motormanagement ein, um so ein Schleudern zu verhindern.

Diese angeführten Beispiele sind sicherheitskritische bzw. life-critical Softwaresysteme, wobei dagegen der Einsatz von einer elektrischen Versperreinrichtung (Anstelle eines Schlüssels) für das Automobil unter die Komfortsysteme fällt.

Die Liste der verschiedenen Software Systeme im Automobil ist damit aber noch lange nicht vollständig,

doch die Aufzählung aller vorhandenen Einrichtungen würde hier den Rahmen weit sprengen und soll nicht Thema dieser Arbeit sein.

Der Einsatz der oben genannten Software-Techniken kann in der Praxis jedoch oft nicht nur positive Auswirkungen haben. Abgesehen von den Fehlfunktionen, die Aufgrund der hohen Komplexität auftreten können, ist bei diesem Softwareumfang auch die Gefahr eines gesamten Ausfalls ein ständiger Begleiter.

3. Qualitätssicherung der Software im Automobil

„Die Software-Qualitätssicherung (SQS) soll sicherstellen, dass die Ziele der Unternehmensführung bezüglich der Qualität der vom Unternehmen produzierten Software erreicht werden." [12]

Dies bedeutet nichts anderes, als dass sich die Automobilhersteller in Punkto Qualität keine „Ausrutscher" leisten dürfen. Das Problem in diesem Sektor ist meist die Konkurrenz und so entscheidet vor allem Preisgünstigkeit und Schnelligkeit, die jedoch bei „life-critical" Software nie im Vordergrund stehen darf, über den Markterfolg eines neuen Automobils. Durch die wachsenden Kundenansprüche stehen Automobilbauer in dem Zwang, schnell geeignete Autos auszuliefern um bei dem wirtschaftlichen Wettstreit nicht als Verlierer hervor zu gehen. Daraus resultiert, dass Entwickler oft gezwungen sind, ein Produkt vorzeitig frei zu geben ohne Praxistests oder Einzelüberprüfungen der Bauteile durchzuführen und so gravierende Mängel auftreten können und die Qualität darunter leidet. Die dabei entstandene Software kann folglich durchtränkt von Fehlern sein was zur Folge hat, dass alle Bemühungen des Unternehmens, eine marktführende Stellung einzunehmen, zunichte gemacht werden.

Doch die Qualität und die Sicherung der Software sind nicht nur aus diesem kapital-schaffendem Grund essentiell. Durch die Einbettung sicherheitskritischer SW ist es zunehmend wichtig, die Qualität dieser zu gewährleisten, um die Fahrzeuginsassen nicht in Gefahrensituationen zu bringen bzw. sie aus solchen Zuständen wieder herausholen zu können. Die serienmäßige Herstellung von Automobilen mit software- und elektronisch gesteuerten Elementen erfordert eine optimale Funktionsweise dieser Bestandteile, um das Risiko eines eventuell lebensbedrohlichen Versagens zu minimieren.

3.1. Qualitätsmerkmale

Aufgrund der leider immer wieder auftretenden Fehlfunktionen im PKW kristallisierten sich die wichtigsten Qualitätsmerkmale und Anforderungen an die Software heraus. Markante zu erfüllende funktionale Eigenschaften sind nicht nur in der Automobilbranche die….

- Zuverlässigkeit,
- Verfügbarkeit,
- Sicherheit,
- Funktionserfüllung,
- Leistung, sprich die Fähigkeit von schwachen Prozessoren viel ausführen zu können,
- Reaktionszeit bezüglich der Echtzeiteinflüsse,
- Benutzerfreundlichkeit,
- Wartungsfreundlichkeit,
- Übertragbarkeit, die an das SW-Gesamtprodukt gestellt werden, um ein Versagen so gut es geht zu unterbinden.

Software ist allerdings ein immaterielles und nicht physikalisches Produkt, wie es der Rest des Automobils ist und so mag es zunächst den Anschein haben, dass durch die schnelle Modifikation ein Vorteil entsteht. Dies kann sich aber rasch ins Negative wenden, wenn z.B. kurzfristige Änderungen vorgenommen werden müssen und so ebenfalls eine gesamte Änderung der Zielplattform, also des Prozessors und der restlichen Automobil-Hardware erforderlich wird. Daher ist das Merkmal der Änderbarkeit der SW ebenfalls entscheidend für die Qualität.

Der rapide Anstieg der Softwarekomplexität sowie der Zeit- und Preisdruck bei der Entwicklung wirkt sich oft negativ auf die Produktqualität aus und so wächst die Herausforderung an die Ingenieure, die Qualität zu sichern, was jedoch meist schon daran scheitert, dass SW-Qualität schwer messbar und definierbar ist.

3.2. Probleme bei mangelnder Qualität

„Der steigende Anteil der Elektronik in den Automobilen und dessen Komplexität schlagen sich bereits in der Pannenstatistik nieder. War nach Zahlen des ADAC 1998 die Elektrik und Elektronik noch in 45,2 Prozent aller Pannen die Fehlerursache, so war sie 2001 schon zu 49,7 Prozent der Pannen das Problem. Mithin war bei jedem zweiten liegen gebliebenen Auto die Elektronik schuld." [15]

Und die neuesten Zahlen, die der ADAC bei einer Fachtagung im Jahr 2003 repräsentierte, sprechen noch eindeutiger über die „unreife Software". So sollen bereits 55% der Pannen ihre Ursache in der fehlerhaften Elektronik und SW haben.

In der letzten Zeit ist auch die Anzahl der Rückrufaktionen von Automobilherstellern auffällig gestiegen. 10% der auftretenden Fehlfunktionen haben ihren Ursprung in der eingebetteten Software und Elektronik, wobei diese Prozentzahl jedoch noch steigen wird, da auch der Software-Anteil selbst im wachsen ist. Immer mehr Fehler treten in den eingebauten Systemen auf und verursachen so horrende Kosten. Der Grund dafür ist nicht schwer zu finden. In punkto Sorgfalt und Gründlichkeit ist der Ruf von Ingenieuren in dieser Branche noch heute besser, als der der Programmierer, was gerade durch solche fehlerhaften Modellreihen noch unterstrichen wird.



Abbildung 2: Anzahl der Rückrufaktionen [25]

Um nur einige Beispiele aufzuzeigen, die von Fachzeitschriften wie Autotouring oder auto-motor-und-sport in den letzten Jahren veröffentlich wurden: *Renault* musste rund 265.000 „Kangoo" wegen Probleme bei dem Bord-Rechner zurückrufen, da dieser unbeabsichtigt den Airbag öffnete. Und auch die *DaimlerChrysler Corporation* musste Schäden in Milliardenhöhe zahlen, als Fahrzeugeigentümer in den USA sie wegen defekter Airbags verklagte. *General Motors* rief 127.000 Sportwagen der Marke Chevrolet Corvette wieder zurück, da die automatischen Lenkschlösser ein Blockieren des Lenkrades verursachen konnten.
Peinlich war des Weiteren ein TDI von *Volkswagen*, der wegen eines Defektes bei der Elektronik nicht abgestellt werden konnte, da die Spritzzufuhr nach dem Abziehen des Zündschlüssels nicht unterbrochen wurde.

Diese Ereignisse, von denen jetzt nur ein Ausschnitt erwähnt wurde, zeugen von unsicherer Software. Die Entwicklung der Elektronik nahm rasant zu, doch die Qualität konnte dabei oft nicht mitgesichert werden. Die Automobilindustrie gilt im Bezug auf die Qualitätssicherung der Produktionsprozesse als Vorreiter für andere Branchen, doch im Bezug auf die Software muss hier noch einiges an Arbeit getan werden.

Um diesem Problem entgegen zu wirken, einigte sich die Automobilbranche auf gemeinsame Programmierstandards wie MISRA.

4. MISRA

„**MISRA** steht für Motor Industry Software Reliability Association mit Sitz in Warwickshire (UK). Misra ist eine Vereinigung verschiedener Organisationen innerhalb des Automotive Sektors, die sich seit 1994 im Rahmen der MIRA (Motor Industry Research Association) gebildet hat. Sie veröffentlicht Programmierrichtlinien mit dem Schwerpunkt auf C zur Best-Practice-Software-Entwicklung und Anwendung für die Automobilbranche. Dem Misra-Standard haben sich inzwischen zahlreiche Automobilhersteller und -Zulieferer angeschlossen. Die Misra-Guidelines können im Internet unter www.misra.org.uk heruntergeladen werden." [4]
Vorteile durch dieses erweiterbare Richtmaß sind, dass ein angesehener Industriestandard und vergleichbare Qualitätszustände erreicht werden können. Dadurch können auch die Verfahren zur Qualitätssicherung, wie zum Beispiel das Test- und Risikomanagement, schnell, aber trotzdem gründlich durchgeführt werden.
Dementsprechend steht MISRA mit seinem Namen für Standards zur Fehlervermeidung bei sicherheitskritischen Systemen und legt somit Qualitätsmaßstäbe für die Industrie fest.

4.1. MISRA Guidelines

Im November 1994 wurden erstmals die MISRA Guidelines von der MIRA veröffentlicht, die speziell für Ingenieure, Manager und sonstige Involvierte der Automobilindustrie als Unterstützung und nicht als Vorschrift gelten sollen.

Hier werden einige Auszüge aus diesen Guidelines vorgestellt, um die Entwicklung von verlässlicher Software so gut es geht zu gewährleisten. (Quelle: www.misra.org.uk)

Einer der ersten relevanten Punkte besteht in der Differenzierung der zu entwickelnden Software und den anderen Formen des Automobil-Ingenieurwesens. So müssen sich die Softwareentwickler zunächst darüber im Klaren sein, dass Software größere Kapazitäten umfasst und so auch mehr Komplexität. Im Gegensatz zu den restlichen HW-Bestandteilen des Automobils ist SW leicht veränderbar und auch unberührbar. Makel in der Designphase haben systematische und nicht zufällig auftretende Softwarefehler zur Folge.
Eine weitere wichtige Unterscheidung für die Entwickler besteht allgemein zwischen der Automobilentwicklung und anderen Industriegebieten bezüglich der Softwareproduktion, da hier das Produktionsvolumen bei weitem größer ist und es sich um datengetriebene Algorithmen handelt. Auch das Fehlermanagement muss bei der Herstellung von Software im Automobil ausgereifter sein, da es hier keine Schulung für die Benutzer – sprich also für den Fahrer gibt.

Des Weiteren werden in den Richtlinien Basis-Prinzipien festgelegt, die für die Softwareentwicklung unumgänglich sind:

- Sicherheit muss vorhanden sein
- Je größer das Risiko, desto größer muß die Menge an Information sein
- SW-Robustheit, Verlässlichkeit und Sicherheit müssen genauso wie die Qualität während der Entwicklung entstehen nicht im nachhinein
- Sicherheit ist wichtiger als Kundenwünsche
- Systemdesign muß Fehler auffangen können
- Sicherheit muss während dem Design, Entwurf und der Herstellung miteinbezogen werden!
- Je später Änderungen vorgenommen werden, desto größer sind die daraus resultierenden Risiken

Diese Voraussetzungen bilden den Grundstock für die weitere Entwicklung, die von MISRA als Software-lifecycle bezeichnet wird und mit einem SW-Entwicklungs-Plan seinen Anfang nimmt. Dieser beinhaltet Spezifikation, Design, Programmierung, Integration der SW in die vorhandene HW, sowie Verifikation und Validation. Darauf folgt der Qualitätsplan, gefolgt vom Sicherheitsplan, der die Aufgabe hat, die Verifikation und Validation während des Projektes zu unterstützen. Den Abschluss bildet der Konfigurations-Management-Plan, der so wie die anderen genannten Pläne vor Anlauf des Projektes erstellt werden muss.

So wurde bereits erwähnt, dass ein Sicherheitsplan vor Beginn des Projektes von großer Bedeutung ist. Die dazugehörige Sicherheitsanalyse bzw. safety analysis dient hierbei unter anderem zur Erhaltung der Vollständigkeit und soll als kontinuierlicher Prozeß während der Requirement-Analyse erfolgen. Weiters sollen durch die Analyse, Gefahren durch Fehler im System vermieden werden (zumindest verringert) und Risiken für jedes einzelne Stadium des Lifecycles gesenkt werden.
„Im April 1998 veröffentlichte die MIRA einen C Regelsatz zur Best-Practice Software Entwicklung innerhalb der Automobilbranche. Diese beinhaltet 93 Regeln und 34 Richtlinien zur Erreichung hoher Automatisierbarkeit für sicherheitskritische Anwendungen. Mit der Vorgabe von SIL3 (Safety Integrity Level nach IEC 61508) kann unter anderem Konsistenz gewährleistet werden. So konnte ein portabler Code für embedded Systems geschaffen werden, der öffentlich zugänglich ist.“ [14]

Das hier angesprochene SIL stellt für Anlagen und Prozesse einen internationalen Sicherheitsstandard dar. Diese Risiko- bzw. Sicherheitsanalyse ermöglicht die Festlegung des so genannten **Safety Integrity Level** nach IEC 61508 (International Electrotechnical Commission). Dieses wird in 4 Levels eingeteilt, die jeweils das Schadensausmaß von auftretenden Fehlern bestimmen:

- Level 4: Unkontrollierbar - Fehler deren Effekte nicht kontrollierbar sind
- Level 3: Schwer kontrollierbar - Effekte normalerweise vom Benutzer nicht kontrollierbar aber könnten durch Spezialisten kontrolliert werden
- Level 2: Lähmend - Fehler meist leicht kontrollierbar durch Spezialisten
- Level 1: Ablenkend - ergibt funktionsbedingte Limitationen, kann aber durch Benutzer kontrolliert werden

Abschließend folgt das Level 0, das jedoch eine reine Beeinträchtigung zur Folge hat. Die auftretenden Fehler gefährden die Sicherheit nicht – hier steht lediglich die Kundenzufriedenheit im Mittelpunkt.

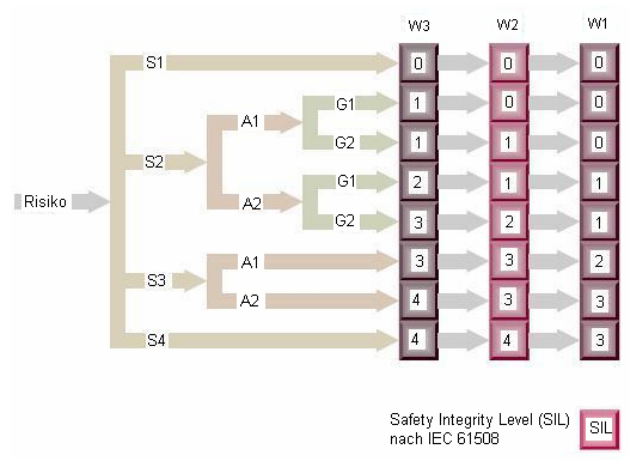


Abbildung 3: Risikoanalyse zur Bestimmung der SIL [29]

Risikoparameter:

S Schadensausmaß
S1 leichte Verletzungen einer Person; kleinere schädliche Umwelteinflüsse
S2 schwere irreversible Verletzungen einer oder mehrerer Personen oder Tod einer Person; vorübergehende größere schädliche Umwelteinflüsse
S3 Tod mehrerer Personen; lang andauernde größere schädliche Umwelteinflüsse
S4 katastrophale Auswirkungen; sehr viele Tote
A Aufenthaltsdauer
A1 selten bis öfter
A2 häufig bis dauernd
G Gefahrenabwendung
G1 möglich unter bestimmten Bedingungen
G2 kaum möglich
W Eintrittswahrscheinlichkeit des unerwünschten Ereignisses
W1 sehr gering
W2 gering
W3 relativ hoch

Eine der wichtigsten Rollen in der Festlegung des Safety Integrity Levels übernimmt der Mensch. Dieser hat den größten Einfluss über die Software während der Fahrt und liefert wichtige Faktoren für die Sicherheitsanalyse, die unbedingt miteinbezogen werden müssen:

- Menschliche Reaktionszeit
- Das Erkennen einer gefährlichen Situation
- Aufmerksamkeit
- Die Erfahrung des Fahrers
- Risikobereitschaft
- Umsetzungsfähigkeit des Fahrers und Handhabung des Systems
- Arbeitsbelastung des Fahrers besonders im Moment des auftretenden Fehlers
- Allgemeine körperliche und geistige Verfassung

Wie in dem letzten Zitat erwähnt, ist es also durch den MISRA Standard möglich geworden, das Sicherheitslevel 3 nicht zu überschreiten und man kann somit gravierende Unfälle mit großen Verlusten vermeiden.

Ein weiterer Lösungsansatz der von MISRA geboten wird, ist der Vorschlag zum Einsatz eines Überwachers bzw. eines Einschätzers, der als unbeteiligter Dritter die Rolle eines Beobachters einnehmen soll. Dieser hat die Aufgabe den Kunden und seine Interessen zu vertreten, das Projekt zu überwachen sowie Fehler und Risiken einzuschätzen. Der Grundgedanke besteht darin, eine gewisse Unabhängigkeit zu erreichen, die nicht nur durch eine andere externe Person, sondern auch durch andere Abteilungen, Sektionen oder Firmen erreicht werden soll. An vorderster Stelle sollte jedoch die Zusammenarbeit der einzelnen Gruppen stehen, um Probleme wie „Betriebsblindheit" vermeiden zu können.

In den MISRA Richtlinien werden auch Empfehlungen für den Gebrauch von **„on-board diagnostics"** ausgesprochen, die die Sicherheit des Fahrzeuges in bestimmten Situationen gewährleisten sollen.

„On-Board-Diagnosesystem (OBD-System) bezeichnet ein an Bord des Fahrzeugs installiertes Diagnosesystem für die Emissionsüberwachung, das in der Lage sein muss, mit Hilfe rechnergespeicherter Fehlercodes Fehlfunktionen und deren wahrscheinliche Ursachen anzuzeigen." [7]
Diese OBD-Systeme liefern Informationen zur Fehleridentifikation und sollen den Kunden so lange es geht mobil halten, indem sie drei verschiedene Arten von Fehlern vermeiden sollen:

- Nichtexistierende oder unkorrekte Sensorensignale
- Bestimmte „SW-Komponenten", die nicht wie erwartet handeln
- Prozesse des Hostsystems die nicht funktionieren, aber deren Auslöser nicht im Prozessor selbst liegen

Wurde nun einer dieser Fehler während des Betriebes gefunden, so löst das Diagnosesystem folgende

Reaktionen aus: Der Fahrer wird gewarnt, Fehlerdaten werden gespeichert, es wird auf „Sparflamme“ geschaltet (bis zur Reparatur) und lokale Fehlermeldungen werden ausgegeben. Wichtig bei diesen Aktionen ist für das System die Unterscheidung zwischen schweren und kleineren Fehlern, deren Relation vom Entwickler festgelegt werden muss.
„On-board-diagnostics“ sollen, laut MISRA, nachträglich in das Gesamtsystem eingebaut, aber auf keinen Fall vergessen werden!

Bei der Programmierung selbst ist besonders darauf zu achten, welche Sprache man verwendet. Durch die Schwierigkeit, Maschinencodes nachzuvollziehen und diese dynamisch zu verändern, haben sich in dem Sektor der Automobilindustrie die Programmiersprachen C und C++ sehr bewährt. Durch die Projektstandards sollen hier verschiedene Richtlinien vorgegeben werden für:

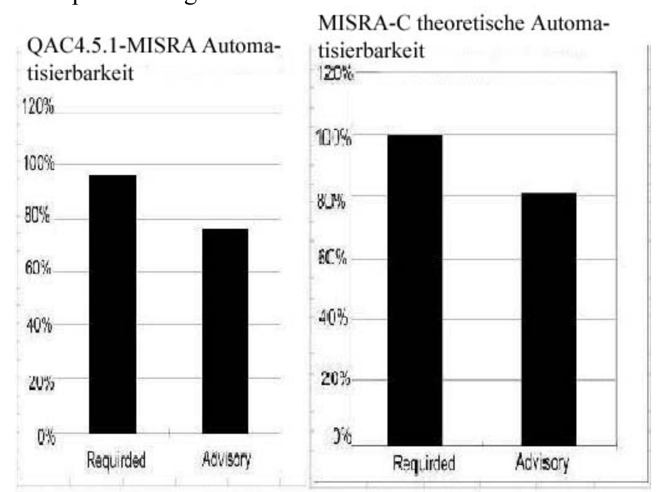
- das source code layout
- den Dokumentationsstil
- den Sprachgebrauch der Programmierung
- die Compilerverwendung
- die Designanwendung

Der Vorteil des Standards besteht vor allem darin, dem Programm eine bessere Struktur geben zu können und dadurch das Auftreten von Fehler zu verringern.

4.2. Pro und Contra

Die Entwicklungsprogramme **QA C** bzw. **QA C++** (quality assurance), die von MISRA zur Fertigung und Erweiterung vorgeschlagen werden, können einen Automatisierungsgrad von bis zu 95% erreichen und ermöglichen es unter anderem, Software-Metriken zu berechnen und auch graphisch darzustellen. Das weltweit führende Analyse Werkzeug erkennt Implementierungsfehler und Inkonsistenzen im Source-Code und bewerkstelligt frühzeitige Verbesserungen um so spätere Korrekturen und deren horrende Kosten zu vermeiden. Mit Hilfe der zur Verfügung gestellten Richtlinien kann die Code-Review-Zeit und Testzeit verkürzt, Komplexitäten limitiert und der Code-Reuse unterstützt werden um so die Produktivität und Qualität zu steigern. Bei der Verwendung dieser computerbasierten Modellierungstools ist jedoch besonders auf das Data Dictionary zu achten, das für den korrekten Einsatz unumgänglich ist.

Um die Effizienz der Verwendung von MISRA QA C/C++ verständlicher zu machen, soll folgende Graphik beitragen.



Die empfohlene und erzielte Automatisierbarkeit von Code-Inspektionen nach MISRA-Standard

Die theoretisch erreichbare und tatsächlich erzielte Inspektionsqualität mit QA-C

Abbildung 4: Automatisierbarkeit [4]

Daraus ist ersichtlich, wie effektiv die Entwicklung dieses Tools war und welche Vorteile sich für die Automobilhersteller zur Qualitätssicherung der eingebetteten Software ergeben.

Dennoch sind die von MISRA angebotenen Programmierstandards nicht nur von Nutzen.

„Die ursprüngliche Stärke der modularen Programmierung in C oder C++, nämlich die Portierbarkeit und Wiederverwendbarkeit, hat dazu geführt, dass sich die meisten Projekte vom ursprünglichen Designziel entfernt haben. Das Übernehmen von bestehenden Modulen, das Hinzufügen von Features, was alles auch fast immer unter Zeitdruck geschieht, kann eine Applikation bis zur totalen Unübersichtlichkeit wachsen lassen, wo eigentlich eine Überarbeitung des Designs nötig wäre.“ [23]

Oft fehlt es auch den Entwicklern an Toleranz und durch fehlende Überprüfung der gegebenen Metriken kann die Hilfestellung unbedeutend werden.

5. Was bringt die Zukunft?

Tür und Tor öffnen sich in der heutigen Zeit für einfallsreiche Entwickler. Die Menschen sind offen für alles, das ihr Leben erleichtern oder erweitern kann und dies ermöglicht es „großen Geistern“, sich in ihrem Ideenreichtum auszutoben.

In der im Mai 2004 erschienen Ausgabe der Autotouring Zeitschrift wird ein verheißungsvoller Einblick in das Auto von Morgen gegeben. So sollen die Elektronik und Software fast die gesamte Kontrolle über das Fahrzeug übernehmen.

- Fahrspuren-Erkennung: Erkennt bei vorhandenen Begrenzungslinien die Position innerhalb der Fahrspur und warnt bei Gefahr, von der Straße abzukommen.
- Umfeldwahrnehmung: Verkehrszeichen (Stopp, Tempolimits, etc.) und Hindernisse werden auf die Scheibe projiziert, Maßnahmen eingeleitet.
- mechatronische Türgriffe: Das Aus für die Schnalle: Wenn man sich einer bestimmten Fläche nähert oder sie berührt, öffnet die Tür von selbst.
- Totwinkelüberwachung: Radar-Sensoren an Spiegeln und Rädern überwachen den toten Winkel in einer Distanz von 0,5 bis 40 Metern, schlagen im Ernstfall Alarm.
- Fahrdynamik-Regelung: Heute nur in Grenz-Situationen, bald jedoch ständige elektronische Antriebs-, Brems-, Lenk- und Fahrwerks-Regelung.
- Drive-by-Wire: Mechanische Elemente wie Lenkgestänge und Gas-Seil werden durch Datenkabel und Elektromotoren ersetzt.
- Aufmerksamkeitskontrolle: Die Cockpit-Kamera überwacht den Lidschlag, die Software berechnet das Sekundenschlaf-Risiko, weist auf Ruhepausen hin.
- Anti-Crash-Komplex: Heute noch (simple) Abstandskontrolle per Radar, bald schon exakte Laser-Vermessung mit Software für automatisches Bremsen und Ausweichen.
- Innenluftmanagement: Eine Kombination aus Luftgüte- Sensoren und Filtern eliminiert Schadstoffe. Das Lieblingsparfum kommt aus dem Zerstäuber.
- Telematik-Box: Ständige Verbindung zu Mobilfunk-Unternehmen, die daraus Verkehrsfluss und Stau-Daten errechnen.
- LED-Frontscheinwerfer: Leuchtdioden ersetzen langfristig die gewohnten Scheinwerfer. Vorteil: zuverlässiger & sparsamer.
- Sicherheits-Programm: Wenn Radar, Laser und Sensoren echte Unfallgefahr wittern, fahren die Sitze automatisch in Idealposition, und die Gurte werden gestrafft.
- Automatische Notbremse: Verschärftes Abstandsradar, das eine Vollbremsung auslöst, wenn ein Hindernis im Weg steht. Ziel: Unfallfolgen sollen vermindert werden.
- Nachtsichtsystem: Fernlicht ohne Blendung: Wärmebild-Kamera erkennt Hindernisse, Bilder davon werden auf die Scheibe projiziert. [26]

Diese Zukunftsvisionen repräsentieren die neuen Sinnesorgane des Automobils und lassen so den menschlichen Fahrer fast überflüssig werden.

Dies mag sehr viel versprechend klingen, aber die traurige Wahrheit dahinter ist, dass je mehr Innovationen umgesetzt werden, desto mehr Fehlfunktionen werden auftreten und so eventuell mehr zu einem Laster werden, als zu einer Unterstützung. So muss sich das Augenmerk in der kommenden Zeit nicht nur auf die neuwertigen Entwicklungen richten, sondern vor allem auch auf die Sicherung der Qualität dieser neuen Software.

6. Resümee

Der Anteil der im Automobil verwendeten Software wird insofern in den nächsten Jahren enorm anwachsen – mit ihm die Notwendigkeit die korrekte Funktionsweise, die Verlässlichkeit und die Qualität mitzusichern. Dieses Gebiet der Informatik hat demnach eine große Zukunft und einen ebenso großen Bedarf an Experten, die sich diesem Thema widmen.

Heutzutage verlassen sich die Hersteller in der Automobilindustrie nicht mehr nur auf die Kollegen der Informatik, sondern erstreben selbst die Kontrolle der Software im PKW zu erlangen. Ein Beispiel dafür ist BMW, das vor einiger Zeit eigens ein Tochterunternehmen (BMW CAR IT) gründete, um die Qualität der eingebetteten, selbst hergestellten Software zu steuern und zu kontrollieren, um eine eigene Qualitätspolitik betreiben zu können.

Daraus entstehen eine Vielzahl von Regeln, Verfahren und Vorschriften zusätzlich zu denen, die durch MISRA vorgegeben werden, und erlauben es so der Qualitätssicherung, sich auf die Überwachung des Entstehungsprozesses der Software zu konzentrieren.

Die Software-Qualitätssicherung liegt jedoch nicht nur in der Hand der Entwickler. Um die Sicherheit zu gewährleisten, ist es besonders notwendig bei dem Entstehungsprozess die Zusammenarbeit nicht nur zwischen den Programmierern und Herstellern vertiefend zu betreiben, sondern auch zwischen den Herstellern und Zulieferer. Angefangen von der

Anforderungsanalyse, den einzelnen Entwürfen und der Implementierung bis hin zu den unterschiedlichen Tests muss die Kooperation und die Einarbeitung der Qualität für die Softwarekomponenten im Automobil und deren Sicherung auf einer gemeinsamen Basis erfolgen. Nur durch ständigen Kontakt, gut funktionierendes Teamwork und gegenseitigen Wissensaustausch erhöht sich die Sicherheit der SW und gleichzeitig deren Qualität, da diese nicht allein in der Designphase oder alleine in der Testphase erreicht werden kann. Durch enge Zusammenarbeit sowie deutlich festgelegte Schnittstellen und Verantwortlichkeitsbereiche wird es möglich in jeder Phase der Herstellung eine Qualitätsüberprüfung entweder durch externe oder interne Fachleute durchzuführen.

So lässt sich erkennen, dass die Software-Qualitätssicherung ein großes Gebiet in der Automobilindustrie einnimmt und ein neues Denken von Entwicklern verlangt, da diese sich nicht nur mehr auf ihre „hardware" im Auto versteifen dürfen. Durch den hohen Prozentsatz der verwendeten Software ist es erforderlich, alle Abteilungen der Unternehmen für die Zusammenarbeit zu motivieren um das gesamte Gedankengut aller Beteiligten einfließen lassen zu können. Der hierbei entstehende Aufwand ist bei Weitem gerechtfertig, wenn dadurch Menschenleben gerettet werden können.

Zum Abschluss möchte ich noch ein Zitat verwenden, das ich bei meinen Recherchen für diese Arbeit gefunden habe und das meiner Meinung nach vieles über die Verwendung von Software in der Automobilindustrie aussagt:

"Weil Fahren ohne Software hard-ware" [10]

6. Referenzen

[1] http://www.all4engineers.com/ Datum: April – Mai, 2004

[2] http://www.oeamtc.at/ Datum: April – Mai, 2004

[3] Auto Touring – Clubmagazin des Öamtc; Ausgabe April 4/2004 (zu finden u.a unter http://www.oeamtc.at/ Datum: April – Mai, 2004)

[4] Zeitschrift AUTO&ELEKTRONIK; Ausgabe April 4/2002 (zu finden auch unter http://www.all-electronics.de/article/013013001/9c915fb430e.html Datum: April, 2004)

[5] Prof. Dr. M. Broy, Dr. M. von der Beeck, I. Krüger – TU München / SOFTBED: Problemanalyse für ein Großverbundprojekt; "Systemtechnik Automobil - Software für eingebettete Systeme"; 12. März 1998 (zu finden unter http://www.bmbf.de/pub/softbed.pdf Datum: April – Mai, 2004)

[6] P. Kleiner: „Die Entwicklung von Software für den PKW", ATZ 10/2003 Jahrgang 105 (zu finden unter http://www.all4engineers.com Datum: April – Mai, 2004)

[7] http://mitglied.lycos.de/Autoelektrik/Zho.htm Datum: Mai, 2004

[8] http://www.sqs.de Datum: April – Mai, 2004

[9]http://www.automobil-forum.de/themen/00136/index.php Datum: April – Mai, 2004

[10] Dr. U. Weinmann: „Anforderungen und Chancen automobilgerechter Software-Entwicklung"; BMW Car IT GmbH, München Juni 2002

[11] http://www.programmingresearch.com Datum: April – Mai, 2004

[12] H.Trauboth: „Software Qualitätssicherung" Konstruktive und analytische Maßnahmen; Handbuch der Informatik; Oldenburg 1993, Bd 5.2

[13] G. E. Thaller: „Software-Qualität" - Entwicklung Test Sicherung; Informatik Management; Sybex Düsseldorf, 1990

[14] A. Sczepansky, QA Systems GmbH: „MISRA Programmierstandard", Qualitätssicherung in der Praxis – Für und Wider Programmierstandards (zu finden unter http://www.weltwissen24.de/fsqrs/termine/020606/MISRA.pdf Datum: April – Mai, 2004)

[15] http://www.eetimes.de/at/news/OEG20040216S0008 Datum: Mai, 2004

[16] http://adac.de Datum: April – Mai, 2004

[17] http://www.auto-motor-und-sport.de Datum: April – Mai, 2004

[18] M. Broy, „ Software im Automobil: Potentiale, Herausforderungen, Trends" http://www.bmw-carit.de/gi/praesentationen/GI_2003_

Praesentation_Software_im_Automobil.pdf Datum: April – Mai, 2004

[19] http://www.all-electronics.de Datum: April – Mai, 2004

[20] http://www.bmw-carit.de Datum: April – Mai, 2004

[21] http://www.qa-systems.de/ Datum: April – Mai, 2004

[22]http://www.telematik-institut.org/presse_und_medien/pressemitteilungen/2002/pm41_02.html Datum: April – Mai, 2004

[23] http://www.elektronikpraxis.de/fachartikel/druck/ep_fachartikel_druck_561630.html Datum: April – Mai, 2004

[24] T. Jungmann, „Auto erkennt Handschrift des Fahrers“ (zu finden unter http://www.all4engineers.com Datum: April – Mai, 2004)

[25] http://www.ttatwest.net/porttal/presse/wiwo2k20919.html Datum: April – Mai, 2004

[26] Auto Touring – Clubmagazin des Öamtc; Ausgabe Mai 5/2004 (zu finden u.a unter http://www.oeamtc.at/ Datum: April – Mai, 2004)

[27] MISRA Guidelines zu finden unter: www.misra.org.uk Datum: Mai 2004

[28] http://de.encarta.msn.com Datum: April – Mai, 2004

[29] M. Kurzmann, Bosch Engineering GmbH: „Testen leicht gemacht“ (zu finden unter: http://en.etasgroup.com/downloads/rt/rt_2003_02_36_en.pdf Datum: Mai, 2004)

Aufwandsschätzung am Beispiel COCOMO II

Daniela ESBERGER
0160072
daniela.esberger@edu.uni-klu.ac.at

Abstract

Nur allzu oft wird aus vermeintlich logischen Gründen wie beispielsweise Zeitersparnis auf eine sorgfältige Aufwandsschätzung verzichtet. Dabei bildet sie einen Grundpfeiler des Projektes und sollte daher auch ihrem Stellenwert entsprechend behandelt werden. Im Speziellen bei Projekten für externe Auftraggeber bildet die Aufwandsschätzung die Basis jeder Angebotserstellung, aber auch bei internen Projekten sollte sie nicht vernachlässigt werden. Oft wird eine Schätzung nur zu Beginn durchgeführt und auf eine regelmäßige Kontrolle und Beobachtung des Verlaufes, um gegebenenfalls vorbeugend einzugreifen, vergessen. Im Folgenden soll das Thema Aufwandsschätzung näher beleuchtet werden und mit COCOMO II dem Leser ein Werkzeug zur praktischen Anwendung geboten werden.

1. Aufwandsschätzung

"Was man nicht misst, das kann man nicht steuern." Dieser Satz von Tom de Marco trifft den Nagel auf den Kopf. Zu Beginn eines Softwareprojektes sind oft Fragen über den Arbeitsaufwand, die Dauer, die Kosten und die Anzahl der Mitarbeiter, die für die Durchführung des Projektes benötigt werden, durch den Projektleiter zu klären. Obwohl zu diesem Zeitpunkt genaue Informationen fehlen, sollte dennoch eine möglichst präzise Schätzung vorgenommen werden, da beispielsweise aufgrund dieser Daten ein Angebot erstellt wird und dies weder zu hoch noch zu niedrig ausfallen darf. Dies stellt natürlich ein erhebliches Problem dar, das mit Hilfe der verschiedenen Methoden der Aufwandsschätzung (siehe Kapitel 1.2) gelöst werden kann. Aufwandsschätzung ist keine einmalige Aufgabe bei der Projektplanung, sondern sollte regelmäßig durchgeführt werden um Abweichungen zu erkennen und um frühzeitig entsprechende Maßnahmen zu ergreifen. [SOMM01]

1.1. Kostenarten

Bei Softwareprojekten unterscheidet man drei Arten von Kosten [SOMM01]:

- Hardware- und Softwarekosten
- Reise- und Schulungskosten
- Personalkosten

Die ersten beiden Punkte sind im Vergleich zu den Personalkosten nahezu zu vernachlässigen. Zwar können speziell die Reisekosten bei Projekten, die an verschiedenen Standorten realisiert werden, ein erhebliches Ausmaß annehmen. Sie sind dennoch meist eher gering. Außerdem kann durch moderne Kommunikationsformen (zB E-Mail, Video- und Telefonkonferenzen) diesen Kosten teilweise entgegengewirkt werden.

Der Personalaufwand hingegen ist der wichtigste und größte Kostenfaktor. Aus diesem Grund wird der errechnete Aufwand des Projektes in der Regel in Personenmonaten angegeben. Dies führt wiederum dazu, dass die Aufwandsschätzung gemeinsam mit der Zeitplanung erfolgt.

Bei den Personalkosten handelt es sich nicht nur um die Entlohnung der beteiligten Programmierer, sondern auch andere Kosten werden in diesen Satz eingerechnet. Dies sind unter anderem:

- Kosten für die Bereitstellung, Beheizung und Beleuchtung der Büroräume,
- Kosten für unterstützendes Personal (zB Buchhalter, Sekretäre, Reinigungspersonal)
- Kosten für Netz und Kommunikation
- Kosten für zentrale Einrichtungen (zB Bibliothek, Aufenthaltsraum, Kaffeeküche)
- Kosten für sozialen Aufwand (zB Sozialversicherung) und Mitarbeiterzuwendungen (zB Erfolgsbeteiligung)

1.2. Methoden

Zur Aufwandsschätzung stehen nach Barry Boehm folgende Methoden zur Verfügung [BOEH81]:

1.2.1. Algorithmische Modelle. Algorithmische Modelle bieten ein oder mehrere Berechnungsverfahren, die eine Vielzahl an Einflussfaktoren berücksichtigen.

Diese Berechnungsverfahren können linear (Faktoren werden addiert), multiplikativ (Faktoren werden multipliziert), analytisch (mathematische Funktion, die nicht linear oder multiplikativ ist), tabellarisch (Werte werden einer Tabelle entnommen) oder zusammengesetzt (Kombination aus den bereits genannten Verfahren) sein.

Im Vergleich zu den anderen Arten haben algorithmische Methoden eine Reihe von Vorteilen. Sie sind objektiv, können nicht durch Wunschvorstellungen oder persönliche Ansichten beeinflusst werden, sind wiederholbar und effizient. Andererseits haben sie auch einige Schwächen. Sie basieren auf früheren Projekten und dabei stellt sich natürlich die Frage, wie repräsentativ diese Projekte für die Zukunft sind. Außerdem können sie keine außerordentlichen Einflüsse berücksichtigen und wie jedes andere Modell auch, sind sie abhängig von der Qualität der Eingabedaten und der Bewertung der Einflussfaktoren.

Die zwei wohl bekanntesten Vertreter dieser Methode sind COCOMO, das in einem späteren Kapitel noch behandelt wird, und die Function Point-Analyse von Allen Albrecht.

1.2.2. Expertenbefragung. Bei dieser Methode werden ein oder mehrere Experten befragt. Sie erstellen ihre Schätzung aufgrund ihrer Erfahrung und ihres Wissens. Man unterscheidet hier also primär zwischen Einzel- und Mehrfachbefragungen. Bei der Einzelbefragung legt ein einziger Experte allein die Schätzwerte hinsichtlich Aufwand, Dauer und Kosten fest. Handelt es sich um einen erfahrenen Experten, der bereits an mehreren ähnlichen Projekten mitgearbeitet hat, dann haben die vorgeschlagenen Schätzwerte im Allgemeinen eine hohe Genauigkeit. Andernfalls können die ermittelten Schätzwerte weit entfernt von den künftigen Istwerten liegen.

Sind mehrere Experten beteiligt, kommt es zu verschiedenen Ergebnissen, die auf einen Endwert gebracht werden müssen. Um ein gemeinsames Resultat zu erreichen, gibt es zwei Möglichkeiten: Einerseits kann ein Mittelwert errechnet werden. Hier können Extremwerte allerdings zu einer Verschleierung führen. Andererseits kann in einer Diskussion, in die alle Experten miteingebunden werden, ein einheitliches Ergebnis gefunden werden. Eine herkömmliche Besprechung, wo alle Experten zusammenkommen und sich auf eine Lösung einigen sollen, hat aber markante Nachteile. Mitglieder können leicht durch wortgewandtere Personen mit mehr Durchsetzungskraft oder Autorität in den Schatten gestellt werden. Um dies zu vermeiden, wurde die Delphi-Methode entwickelt, die in verschiedenen Bereichen einsetzbar ist. Kurz zusammengefasst funktioniert die Delphi-Methode wie folgt: Die Experten bleiben untereinander anonym. Dadurch können persönliche Faktoren größtenteils ausgeschlossen werden. Eine Monitorgruppe holt die Meinungen der Experten ein und versucht durch eine Befragung dieser zu einem Ergebnis zu kommen. Problematisch ist hier allerdings der hohe Zeitbedarf und so ist dies nur bei großen Projekten auch wirklich sinnvoll.

Da die Stärken und Schwächen der Expertenbefragung genau konträr zu denen der algorithmischen Modelle sind, wird diese Methode oft zu deren Unterstützung verwendet. Die Stärken liegen vor allem darin, dass neue Techniken, Architekturen, Anwendungen und außerordentliche Einflüsse berücksichtigt werden können. Der Nachteil dieser Methode liegt in der Subjektivität. Die Schätzung ist nur so gut wie der Experte, wobei hier natürlich immer eine persönliche Note miteinfließt.

1.2.3. Analogiemethode. Diese Methoden versuchen einen Bezug zwischen vergangenen Entwicklungen und der geplanten Entwicklung herzustellen. Man bedient sich hier Erfahrungsdaten aus ein oder mehreren abgeschlossenen Projekten unter der Verwendung von Vergleichskriterien. Die Analogiemethode kann entweder auf das ganze Projekt oder auf Teile dessen angewendet werden. Sie hat vor allem den Vorteil, dass sie auch schon in den Frühphasen eines Projektes eingesetzt werden kann und sich an aktuellen und tatsächlichen Erfahrungswerten orientiert. Andererseits entsteht natürlich das Problem, dass nicht genau geklärt werden kann, inwiefern die Einflussfaktoren von früheren Projekten sich auch tatsächlich auf ein aktuelles auswirken.

1.2.4. Parkinsons Gesetz. Laut dem Gesetz von Parkinson, nimmt die Dauer einer Arbeit immer den Zeitraum an, der ihr zur Verfügung steht. Man geht also von einem festen Zeitrahmen und den Ressourcen (Mitarbeitern), die zur Verfügung stehen, aus. Hat man nun also 3 Monate bis zur Fertigstellung des Auftrages Zeit und verfügt man über 6 Mitarbeiter, so braucht man nach diesem Gesetz 18 Personenmonate. In manchen Fällen hat sich diese Schätzung als sehr genau erwiesen, in anderen wiederum als stark abweichend. Diese Methode wird nicht empfohlen.

1.2.5. Price-To-Win. Hier werden die Projektkosten durch die beim Auftraggeber verfügbaren Mittel bestimmt. Der zu erwartende Aufwand wird durch das Budget des Auftraggebers und nicht durch das Projekt

selbst bestimmt. Hier kommt es oft zu Überschreitungen des Zeitplans und des Budgets, was wiederum zu Unzufriedenheit führt.

Price-To-Win hat eine Vielzahl von Ausschreibungen für viele Softwareunternehmen gewonnen. Allerdings sind viele davon verständlicherweise heute nicht mehr am Markt.

1.2.6. Top-Down. Bei der Top-Down-Methode wird zuerst der Gesamtaufwand für ein Projekt geschätzt und dann wird dieser für die einzelnen Komponenten heruntergebrochen. Dieses Verfahren kann in Kombination mit jeder anderen bisher genannten Methode erfolgen. Vorteilhaft hierbei ist, dass das Projekt als eine Einheit gesehen wird und nicht auf Kosten für Systemintegration, Benutzerhandbuch etc. vergessen wird. Der Nachteil ist allerdings, dass auf Schwierigkeiten in Einzelfällen oft nicht im Vorhinein als solche erkannt werden.

1.2.7. Bottom-Up. Die Bottom-Up-Methode ist das Gegenstück zu Top-Down und kann ebenso in Kombination zu den anderen Methoden angewandt werden. Es wird jede einzelne Komponente geschätzt und zu einem Gesamtaufwand addiert. Die Vor- und Nachteile sind demzufolge gegengleich zu Top-Down.

2. COCOMO II

COCOMO zählt zu den algorithmischen Modellen zur Kosten- und Aufwandsschätzung von Software. COCOMO wurde Ende der siebziger Jahre erschaffen und 1981 von Barry Boehm veröffentlicht.

Barry W. Boehm studierte Mathematik in Harvard und ist heute Professor an der University of Southern California. Den Großteil seiner Arbeiten entwickelte er während seiner Beschäftigung bei TRW, wo er von 1973 bis 1989 Hauptwissenschaftler der Verteidigungssystemgruppe war. Neben COCOMO entwickelte er auch das spiralförmige Modell des Softwareprozesses, die Theorie W (win-win), das TRW Softwareproduktivitätssystem und die Quantensprungumgebung. Sein heutiger Forschungsbereich umfasst Softwareprozessmodellierung, Software Requirements Engineering, Softwarearchitekturen, Softwaremaßsysteme, Kostenmodelle, Softwaretechnikumgebungen und wissensbasierte Softwaretechnik. [TUWI04]

Das COnstructive COst MOdel (COCOMO) basiert auf empirischen Werten, die aus Datensammlungen aus verschiedenen Projekten und Analysen gewonnen wurden. [SOMM01]

COCOMO II ist gut für die Anwendung in Unternehmen geeignet und gehört zu den wenigen Schätzverfahren, die alle Einflüsse (Quantität, Qualität, Projektdauer und Produktivität) berücksichtigen. [KNOE91]

Außerdem spricht für COCOMO, dass es gut dokumentiert ist sowie häufig verwendet und ausgewertet wird. Des Weiteren ist es frei verfügbar und wird von Public-Domain und kommerziellen Werkzeugen unterstützt. [SOMM01]

Aus verschiedenen Gründen, die in Kapitel 2.1 näher ausgeführt werden sollen, kam es zu einer Überarbeitung des ursprünglichen Modells, das heute COCOMO 81 genannt wird, und zur Neuentwicklung und Veröffentlichung im Jahr 2000 von COCOMO II.

2.1. Von COCOMO zu COCOMO II

Mittels empirischen Untersuchungen erkannte Barry Boehm einen funktionalen Zusammenhang zwischen Systemgröße (nach Anzahl der Codezeilen) und dem Erstellungsaufwand. Da eine derartige Gleichung keine zufriedenstellenden Ergebnisse lieferte, berücksichtigte er zusätzlich Einflussfaktoren für Qualitätsanforderungen und Produktivität. Auf diesem Wege entstand COCOMO. [KNOE91]

Je nach Entwicklungskomplexität und -schwierigkeit gibt es bei COCOMO 81 [BOEH00], wie das ursprüngliche Modell heute genannt wird, drei Berechnungen. Man unterscheidet hier zwischen einfach (basic), mittel (intermediate) und detailliert (detailed). Ausgangspunkt für die Schätzung ist die Größe des Projektes, also die Schätzung der Anzahl der benötigten Codezeilen in KDSI (kilo delivered source instruction). Mit Hilfe einer Formel werden dann die Personenmonate errechnet. Eine Erhöhung der Qualität der Schätzung kann mittels Multiplikation vorbestimmter Einflussfaktoren, deren Werte aus einer Tabelle zu entnehmen sind, erfolgen.

Seit Entwicklung von COCOMO 81 haben sich im Bereich Softwareentwicklung drastische Veränderungen ergeben, die eine Überarbeitung des ursprünglichen Modells unumgänglich machten. COCOMO 81 basiert auf dem Wasserfallmodell, was heute als überholt und veraltert gilt, und auf der Annahme, dass Software immer neu entwickelt wird. Mittlerweile spricht man von komponentenbasierter Softwareentwicklung, wo bereits fertige Komponenten zusammengefügt werden und es so zu einer Wiederverwendung der Software kommt. Des Weiteren wurde die Entwicklung von Prototypen nicht berücksichtigt. Heute ist dies jedoch eine weit verbreitete Methode. Auch werden viel mehr gekaufte Subsysteme eingesetzt, Programmiersprachen der 4. Generation (4GLs) ver-

wendet, vorhanden Software wird umgestaltet und daraus neue kreiert und häufig wird die Entwicklung durch ein CASE-Werkzeug unterstützt. Um all diese Änderungen auch in der Aufwandsschätzung dementsprechend zu berücksichtigen, überarbeitete Barry Boehm dahingehend das Modell und veröffentlichte im Jahr 2000 COCOMO II. [SOMM01]

2.2. Zielgruppen

Für wen eignet sich überhaupt COCOMO II? Diese Frage soll hier im Folgenden geklärt werden. Man geht hier, wie Abbildung 1 zeigt, von fünf verschiedenen Gruppierungen aus.

Der Großteil wird hier der Endbenutzerprogrammierung zugeordnet. Hier handelt es sich in der Regel um kleine Projekte, wo der Programmierer in vordefinierten Umgebungen (zB Tabellenkalkulation, SQL-Client, Planungssysteme) arbeitet und das Programm auf die Bedürfnisse des Endbenutzers abstimmt. Hier wird aufgrund des geringen Projektumfangs nur eine sehr einfache Aufwandsschätzung benötigt.

Endbenutzerprogrammierung (55 Millionen Programmierer in den USA – 2005)		
Generierung v. Applikationen u. Hilfsmittel für d. Aufbau (0.6 Millionen)	Aufbau v. Applikationen (0.7 Millionen)	Systemintegration (0.7 Millionen)
Infrastruktur (0.75 Millionen)		

Abbildung 1. Bereiche der SW-Entwicklung [BOEH00]

Die mittlere Ebene umfasst folgende drei Untergruppen:

1. *Generierung von Applikationen und Hilfsmittel für den Aufbau*
   Es erfolgt häufig eine Wiederverwendung von Komponenten, aber auch neue Funktionalitäten werden hinzugefügt. Die bekanntesten Firmen in diesem Sektor sind beispielsweise Microsoft, Netscape, Lotus, Novell und Borland.
2. *Aufbau von Applikationen*
   Hier kann keine Wiederverwendung der Komponenten aufgrund von Umfang oder Diversifizierung der Produkte erfolgen. Allerdings können die Komponenten (zB graphische Benutzerschnittstellen) leicht mittels CASE-Werkzeugen erstellt werden.
3. *Systemintegration*
   Man spricht hier von systemnaher („embedded“) Software, die vor allem in Bereichen wie Telekommunikation, Luft- und Raumfahrt, Automobilindustrie sowie im Finanzsektor benötigt wird.

Auf unterster Ebene findet man die Programmierung der Infrastruktur. Typische Bereiche sind beispielsweise Betriebssysteme, Datenbankmanagementsysteme, Benutzerschnittstellenmanagementsysteme und Netzwerksysteme. Bekannte Firmen auf diesem Sektor sind Microsoft, Netscape, Oracle, Sybase, 3Com und Novell. [BOEH00]

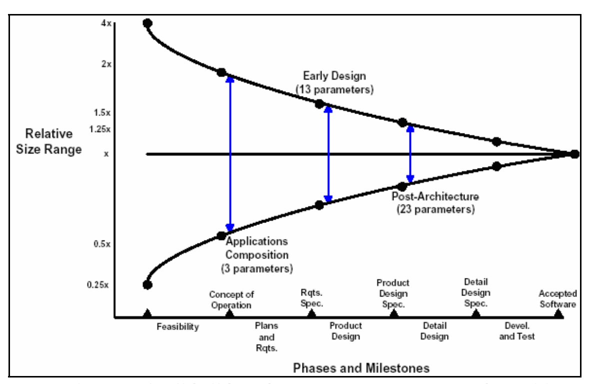


Abbildung 2. COCOMO II Modellphasen [BOEH00]

2.3. Das Modell

COCOMO II ist im Grunde genommen nicht nur ein einzelnes Modell. Je höher die Entwicklungsstufe und das Fortschreiten des Projektes, umso genauer kann auch die Abschätzung erfolgen (siehe Abbildung 2). Daher besteht COCOMO II aus drei Teilmodellen, die sich bezüglich Skalenfaktoren, Aufwandsmultiplikatoren und Modellkonstanten unterscheiden. Hier ein kurzer Überblick:

1. *The Application Composition Model*
   (Die frühe Prototypenstufe)
   Die erste grobe Größenschätzung basiert auf der Methode der Object Points und erfolgt mit einer einfachen Größen-/Produktivitätsformel.
2. *The Early Design Model*
   (Die frühe Entwurfsstufe)
   In dieser Stufe sind bereits alle Systemanforderungen niedergeschrieben und es existiert vielleicht auch schon ein erster Entwurf. Die Schätzung basiert auf Function Points.
3. *The Post-Architecture Model*
   (Die Stufe nach dem Architekturentwurf)
   Die Systemarchitektur wurde bereits entworfen und demzufolge kann eine genauere Bewertung erfolgen. Das Post-Architecture Model verwendet zur Feinabstimmung der Schätzung eine Vielzahl von Multiplikatoren, die u.a. die Fähigkeiten der

Mitarbeiter sowie die Produkt- und Projekteigenschaften ausdrücken. [SOMM01]

2.3.1. The Application Composition Model. Die frühe Prototypenstufe, wie das Application Composition Model auch genannt wird, dient als Unterstützung bei der Schätzung von Prototypen und Projekten, die eine Software hervorbringen sollen, die aus vorhandenen Komponenten zusammengesetzt wird.

In dieser frühen Phase ist es noch sehr schwer, eine genaue Bewertung durchzuführen. Die Basis stellen hier Object Points dar. Sie sind bei Programmiersprachen der 4. Generation (anwendungsbezogene bzw. applikative Sprachen) und anderen vergleichbaren Sprachen eine Alternative zu Function Points, da sie im frühen Stadium noch leichter zu schätzen sind. Bei Object Points handelt es sich nicht um Objektklassen sondern um eine gewichtete Schätzung der folgenden Faktoren:

- *Anzahl der angezeigten Bildschirmmasken*
  Einfache Masken und Dialogfelder gelten als 1 Object Point, komplexere als 2 und sehr komplexe als 3.
- *Anzahl der erzeugten Berichte*
  Einfache Berichte zählen für 2, komplexere für 5 und schwer erzeugbare Berichte für 8 Object Points.
- *Anzahl der Module in Programmiersprachen der 3. Generation (höhere Programmiersprachen)*
  Jedes Modul, das in einer höheren Programmiersprache erzeugt werden muss, zählt für 10 Object Points.

Die errechneten Object Points werden in weiterer Folge durch einen Standardwert für die Produktivität dividiert. Dieser Wert ist abhängig von der Erfahrung und Fertigkeit des Programmierers sowie von den Fähigkeiten der verwendeten CASE-Werkzeuge und lässt sich aus Tabelle 1 ersehen. [BOEH00]

Tabelle 1. Produktivität in Object Points pro Monat

Erfahrungen u. Fähigkeiten d. Entwicklers	sehr gering	gering	mittel	hoch	sehr hoch
Fähigkeiten der CASE-Werkzeuge	sehr gering	gering	mittel	hoch	sehr hoch
PROD (NOP/Monat)	4	7	13	25	50

Da in dieser Phase durchaus bereits erstellte Komponenten wieder verwendet werden, muss auch ein dementsprechender „Reuse-Anteil" (in Prozent – %reuse) miteinbezogen werden. Daher werden die Object Points (OP) um diesen Faktor gewichtet (NOP).

Die Formel zur Berechnung des Aufwandes in Personenmonaten (PM) sieht wie folgt aus [SOMM01]:

$$PM = (OP*(1-\%reuse/100))/PROD$$
bzw.
$$PM = NOP/PROD$$
$$NOP = OP*(1-\%reuse/100)$$

Abbildung 3. Formel: Application Composition Model

2.3.2. The Early Design Model. Dieses Modell wird im Deutschen "Die frühe Entwicklungsstufe" genannt. Die Schätzungen, die in dieser Stufe erfolgen, basieren auf der Standardformel algorithmischer Modelle (siehe Abbildung 4). Das Modell wird, wie sein Name bereits aussagt, in der frühen Phase der Entwicklung, wo noch wenige Informationen über Umfang des Projektes, Zielplattform, Eigenschaften der Personen, die bei diesem Projekt mitarbeiten, und keine Einzelheiten des Entwicklungsprozesses vorliegen, verwendet. Es eignet sich sowohl für die Generierung von Applikationen als auch für die Systemintegration und die Entwicklung von Infrastruktur (siehe Zielgruppen). [BOEH00]

$$PM = A*Größe^{E}*M$$
bzw.
$$PM = A*Größe^{E}*M+PM_{m}$$

$$M = PERS*RCPX*RUSE*PDIF*PREX*FCIL*SCED$$
$$PM_{m} = (ASLOC*(AT/100))/ATPROD$$

Abbildung 4. Formel: Early Design Model

Für den Koeffizient A empfiehlt Barry Boehm in dieser Phase den Erfahrungswert 2,5. Dieser Wert basiert auf der Auswertung der Datenmenge von zahlreichen Projekten. [SOMM01]

Tabelle 2. Umrechnungstabelle FP in SLOC

Sprache	SLOC/UFP	Sprache	SLOC/UFP
4GLs	20	Lisp	64
Assembly	320	Pascal	91
C	128	Perl	27
C++	55	Prolog	64
HTML	15	VB 5.0	29
Java	53	Visual C++	34

Die Größe wird in KSLOC (kilo source lines of code), also die Quellcodezeilen in Tausend, angegeben. Zur Erleichterung kann hier auch die Anzahl der Function Points in der Software geschätzt werden und mit Hilfe von Standardtabellen in SLOC (source lines of code) und in weiterer Folge in KSLOC umgerechnet werden. Diese Standardtabellen geben die Anzahl der

SLOC pro Function Point für jede Programmiersprache an. Tabelle 2 gibt hier die Werte für eine Auswahl der wichtigsten Programmiersprachen an. [BOEH00]

Der Exponent E soll den steigenden Aufwand bei wachsender Projektgröße ausdrücken. Der Wert ist hier nicht genau festgelegt, sondern kann zwischen 1,01 und 1,26 liegen. Abhängig ist die Größe dieser Zahl von der Neuartigkeit des Projekts, der Entwicklungsflexibilität, den verwendeten Prozessen zur Risikolösung, dem Teamzusammenhalt und der Ausgereiftheit des Prozesses. [BOEH00]

Zur Berechnung des Exponenten werden nun die fünf Skalierungsfaktoren (siehe Tabelle 3) nach einer Sechs-Punkte-Skala (5 – sehr gering bis 0 – besonders hoch) bewertet. Die Ergebnisse werden addiert, durch hundert dividiert und zum fixen Wert 1,01 hinzugezählt.

Tabelle 3. Skalierungsfaktoren für den Exponenten

Faktor	**Bemerkungen**
Neuartigkeit	Erfahrung mit dieser Art von Projekten 0 – totale Vertrautheit 5 – keine Erfahrung
Entwicklungsflexibilität	Flexibilität im Entwicklungsprozess 0 – keine Vorgaben der Prozesse 5 – Vorgabe des zu verwendenden Prozess
Architektur/ Risikoauflösung	Umfang der Risikoanalyse 0 – vollständige Analyse 5 – geringe/keine Analyse
Teamzusammenhalt	Vertrautheit und Zusammenarbeit im Team 0 – integriertes und effektives Team 5 – sehr schwierige Interaktion
Ausgereiftheit des Prozesses	Ausgereiftheitsgrad des Prozesses abhängig vom CMM Maturity Questionnaire. Schätzung durch Abzug des Ausgereiftheitsgrades des CMM-Prozesses vom Wert 5.

Der Multiplikator M in der Formel des Early Design Models (siehe Abbildung 4) gibt eine vereinfachte Reihe von Projekt- und Prozessfaktoren wieder. Dazu zählen:

- RCPX – Product Reliability and Complexity (die Produktzuverlässigkeit und –komplexität)
- RUSE – Developed for Reusability (der benötigte Wiederverwendungsgrad)
- PDIF – Platform Difficulty (der Schwierigkeitsgrad der Plattform)
- PERS – Personnel Capability (die Mitarbeiterfähigkeiten)
- PREX – Personnel Experience (die Erfahrung des Personals)
- FCIL – Facilities (die unterstützenden Einrichtungen)
- SCED – Required Development Schedule (der Zeitplan)

Auch diese Faktoren werden mittels einer Skala (siehe Tabelle 4) bewertet. Multipliziert man die Ergebnisse, gelangt man zum Multiplikator M.

Tabelle 4. Projekt- und Prozessfaktoren

	---	--	-	~	+	++	+++
RCPX	0,49	0,60	0,83	1	1,33	1,91	2,72
RUSE			0,95	1	1,07	1,15	1,24
PDIF			0,87	1	1,29	1,81	2,61
PERS	2,12	1,62	1,26	1	0,83	0,63	0,50
PREX	1,59	1,33	1,22	1	0,87	0,74	0,62
FCIL	1,43	1,30	1,10	1	0,87	0,73	0,62
SCED		1,43	1,14	1	1	1	n/a

Alternativ zur oben genannten Bewertung können die Faktoren auch durch eine Kombination der im Post Architecture Model verwendeten Multiplikatoren berechnet werden. [BOEH00]

Wird ein bedeutender Teil des Codes automatisch erstellt, so sollte dies auch berücksichtigt werden. Hier empfiehlt sich ein Formel mit der zusätzlichen Variable PM_m: $PM = A*Größe^E*M+PM_m$ (siehe auch Abbildung 4). Obwohl auch hier meist manuelle Eingaben erforderlich sind, ist der Produktivitätsgrad signifikant höher als bei manuell erstelltem Code.

PM_m berechnet sich mit der folgender Formel: $PM_m = (ASLOC*(AT/100))/ATPROD$ (siehe auch Abbildung 4). ASLOC (automatically generated source lines of code) stellt, wie der Name bereits sagt, die Anzahl der Zeilen dar, die automatisch erstellt wurden. ATPROD steht für den Grad der Produktivität für diese Codeerstellung. Weiters ist ein bestimmter Aufwand für die Herstellung einer Schnittstelle zwischen dem erzeugten Code und dem restlichen System nötig. Da dieser vom prozentuellen Anteil (AT) des automatisch erstellten Codes abhängig ist, ist auch dies zu berücksichtigen. [SOMM01]

2.3.3. The Post-Architecture Model. Das Post-Architecture Model, also die Stufe nach dem Architekturentwurf, ist das detaillierteste der drei Modelle. Es sollte dann verwendet werden, wenn bereits die Softwarearchitektur festgelegt wurde. Einsatz findet das Modell in den Breichen Generierung von Applikationen, Systemintegration und Infrastrukturprogrammierung (siehe Zielgruppen).

Die Schätzung für dieses Modell basiert auf der gleichen Grundformel wie das Early Design Model (siehe Abbildung 4 und Abbildung 5). Da zu diesem späteren Zeitpunkt nun schon mehrere Informationen zur Verfügung stehen, werden statt den sieben Projekt- und Prozessfaktoren nun 17 Einflussgrößen (oft auch „Cost Driver" genannt) verwendet, die durch diesen zusätzlichen Detailliertheitsgrad zu einem genaueren Ergebnis führen sollen. [BOEH00]

$$PM = A * Größe^{E} * M$$

bzw.

$$PM = A * Größe^{E} * M + PM_{m}$$

$$M = \Pi(\text{Cost Driver})$$

$$PM_{m} = (ASLOC * (AT/100))/ATPROD$$

Abbildung 5. Formel: Post-Architecture Model

Die Einflussfaktoren sind vier verschiedenen Gruppen zugeteilt:

- *Produktattribute*
  - RELY – Required Software Reliability (die erfolgreiche Systemzuverlässigkeit)
  - DATA – Data Base Size (die Größe der verwendeten Datenbank)
  - CPLX – Product Complexity (die Komplexität der Systemmodule)
  - RUSE – Required Reusability (der benötigte Wiederverwendungsgrad)
  - DOCU – Documentation match to LC needs (der Umfang der erforderlichen Dokumentation)
- *Computerattribute*
  - TIME – Execution Time Constraint (die Beschränkung der Ausführungszeit)
  - STOR – Main Storage Constraint (die Speicherbeschränkung)
  - PVOL – Platform Volatility (die Unbeständigkeit der Entwicklungsplattform)
- *Personalattribute*
  - ACAP – Analyst Capability (die Fähigkeiten der Projektanalytiker)
  - PCAP – Programmer Capability (die Fähigkeit der Programmierer)
  - PCON – Personal Continuity (die Personalkontinuität)
  - APEX – Applications Experience (die Erfahrungen der Analytiker auf dem Gebiet des Projektes)
  - PEXP – Platform Experience (die Erfahrung der Programmierer auf dem Gebiet des Projektes)
  - LTEX – Language and Tool Experience (die Erfahrung mit der Sprache und den Werkzeugen)
- *Projektattribute*
  - TOOL – Use of Software Tools (die Verwendung von Softwarewerkzeugen)
  - SITE – Multisite Development (die Anzahl der Arbeiten, die an mehreren Stellen ausgeführt werden, und die Kommunikation zwischen diesen Stellen)
  - SCED – Required Development Schedule (die Komprimierung des Entwicklungsplans)

Wie auch beim Early Design Model können diese Größen bewertet werden und ihre Produktsumme ergibt wiederum den Multiplikator M. Die Werte sind folgender Tabelle zu entnehmen [BOEH00]:

Tabelle 5. Cost Driver beim Post-Architecture Model

	--	-	~	+	++	+++
RELY	0,82	0,92	1	1,10	1,26	n/a
DATA	n/a	0,90	1	1,14	1,28	n/a
CPLX	0,73	0,87	1	1,17	1,34	1,74
RUSE	n/a	0,95	1	1,07	1,15	1,24
DOCU	0,81	0,91	1	1,11	1,23	n/a
TIME	n/a	n/a	1	1,11	1,29	1,63
STOR	n/a	n/a	1	1,05	1,17	1,46
PVOL	n/a	0,87	1	1,15	1,30	n/a
ACAP	1,42	1,19	1	0,85	0,71	n/a
PCAP	1,34	1,15	1	0,88	0,76	n/a
PCON	1,29	1,12	1	0,90	0,81	
APEX	1,22	1,10	1	0,88	0,81	n/a
PEXP	1,19	1,09	1	0,91	0,85	n/a
LTEX	1,20	1,09	1	0,91	0,84	
TOOL	1,17	1,09	1	0,90	0,78	n/a
SITE	1,22	1,09	1	0,93	0,86	0,80
SCED	1,43	1,14	1	1	1	n/a

Bei der Größe des Projektes werden zusätzlich zwei wichtige Einflüsse berücksichtigt [SOMM01]:

- *Die Unbeständigkeit der Anforderungen*
  Man geht davon aus, dass eventuell noch Nacharbeiten aufgrund von Änderungen der Systemanforderung zu leisten sind. Diese sollen in ihrem

Zeilenausmaß geschätzt werden und zu der „normalen" Größenschätzung hinzuaddiert werden.

– *Der Umfang einer möglichen Wiederverwendung*
  Bei einem umfangreichen Reuse muss die Zeilenanzahl des tatsächlich entwickelten Quellcodes verändert werden. Nun ist es jedoch so, dass die Kosten für eine Wiederverwendung aufgrund des benötigten Aufwandes zur Findung und Auswahl der Komponenten und deren Schnittstellen sowie den notwendigen Änderungen nicht linear verlaufen. Dies findet in COCOMO II Berücksichtigung. Der Größenaufwand wird durch nachstehende Formel angenähert.

$$ESLOC = ASLOC*(AA+SU+0{,}4DM+0{,}3CM+0{,}3IM)/100$$

Abbildung 6. Formel für Reuse

ESLOC (extension source lines of code) steht für die Zeilenanzahl des neuen Codes, ASLOC (adapted source lines of code) für die änderungsbedürftige Zeilenanzahl des wiederverwendbaren Codes, DM (design modifications) für den prozentuellen Anteil des geänderten Entwurfs, CM (code modifications) für den prozentuellen Anteil des geänderten Codes und IM (modifications for integration) für den prozentuellen Anteil des anfänglich erforderten Aufwandes für die Integration der wiederverwendeten Software. SU (source usability) ist ein Faktor, der auf den Kosten für das Beherrschen der Software basiert. Der Wert kann variabel zwischen 10 und 50 gewählt werden. 10 bedeutet, dass der Code gut geschrieben und objektorientiert ist. Bei 50 handelt es sich um einen sehr komplexen und unstrukturierten Code. Der Faktor AA (application assessment) spiegelt die ursprünglichen Beurteilungskosten für die Entscheidung über die Wiederverwendung der Software wider. Auch hier kann der Wert variabel gewählt werden. Der Wertebereich liegt zwischen 0 und 8.

Wie beim Early Design Model wird ebenso hier für A der Erfahrungswert 2,5 eingesetzt. Auch der Exponent E kalkuliert sich wie bereits im vorigen Modell demonstriert und an der Berechnung von PM_m ändert sich genauso wenig.

2.4. Einsatzgebiete

COCOMO II wurde entwickelt um bei folgenden Entscheidungen und Situation zu helfen [BOEH00]:

- Entscheidungen, die die Investitionen eines Softwareprojektes betreffen,
- Festsetzungen von Projektbudgets und Zeitplänen (Wie viele Mitarbeiter werden in welcher Phase gebraucht? Wie hoch ist der Aufwand zur Erreichung eines bestimmten Meilensteines?),
- Diskussionen über Kosten, Zeitpläne und Leistung,
- Entscheidungen über Risiken,
- Entscheidungen über Fortschritt und Verbesserungen von Software (Reuse, Outsourcing, Tools,…);

2.5. Tools

Es gibt sowohl kostenlose/private Tools als auch kommerzielle Tools, die bei der Aufwandsschätzung mit COCOMO II den Benutzer unterstützen sollen. Hier sollen drei nützliche Tools kurz vorgestellt werden [UNIM04]:

1. *QuickSoft-Calculator*
   Der QuickSoft-Calculator ist zu finden unter: http://www.pm99.de/oldhome/quicksoft/index.htm
   QuickSoft wurde speziell für Studenten von Dr. Siegfried Seibert (Professor an der Fachhochschule Darmstadt) entwickelt um ihnen den praktischen Einsatz zu erleichtern und um das Modell auszuprobieren. Dieses Tool steht in Form einer Web Applikation zur freien Verfügung. Es ist sehr übersichtlich und leicht handhabbar.
2. *Costar*
   Costar ist ein kommerzielles Tool. Die Einzellizenz kostet 1.900 USD, die Unternehmenslizenz liegt bei 25.000 USD. Nähere Informationen zum Produkt sowie eine kostenlos downloadbare Demoversion findet man unter: http://www.softstarsystems.com/index.htm.
3. *Construx Estimate 2.0*
   Construx bietet auf seiner Homepage dieses Tool zum freien Download mit eingeschränkter Lizenz (vorher ist allerdings eine Registrierung notwendig) an. Zu finden ist das Tool unter: http://www.construx.com/resources/estimate/index.php Die bekanntesten Partner sind Dell, Nokia, Microsoft, Intel und General Electric.

2.6. Kritische Betrachtung

Durch die vielen Einflussfaktoren, die berücksichtigt werden müssen, wird die praktische Anwendbarkeit von COCOMO etwas komplex. Da diese Einflussfaktoren die ursprüngliche Schätzung sehr stark beeinflussen können – so können hohe Werte zu einer dreimal so hohen Schätzung führen, niedrige Werte zu

einem Drittel der ursprünglichen Schätzung – ist eine sorgfältige Bewertung umso wichtiger.

Weiters wird kritisiert, dass das Ziel, dass Anwender dieses Modells ihre eigenen Erfahrungen berücksichtigen und die Attributwerte auf ihren historischen Projektdaten basieren, nicht erreicht wird, da in der Regel zu wenige oder gar keine dieser Daten vorhanden sind. Hier könnte man entgegnen, dass COCOMO für den Beginn Standardwerte vorgibt, die erfahrungsgemäß gute Ergebnisse liefern. Ergänzend und zur Verfeinerung können dann über den Zeitverlauf gewonnene Erfahrungen mitberücksichtigt werden. [SOMM01]

Überdies gelten für COCOMO natürlich auch die allgemeinen Kritikpunkte an algorithmischen Schätzverfahren (siehe Kapitel 1.2).

Abschließend ist an dieser Stelle anzumerken, dass keine Methode frei von Kritik ist und COCOMO gegenüber den anderen Verfahren viele Vorteile bietet.

3. Resümee

Zusammenfassend soll hier noch einmal verdeutlicht werden, wie wichtig Aufwandsschätzung für jedes Projekt und besonders auch für Softwareentwicklungsprojekte ist. Insbesondere bei solchen Projekten fällt die richtige Schätzung oft schwer, da viele oft unvorhersehbare Faktoren hier mitspielen und der Fortschritt und Erfolg von Können und Kreativität der Programmierer abhängig ist. Dennoch sollte trotz der Vielzahl an Unsicherheiten nicht auf eine Aufwandsschätzung verzichtet werden. Diese Schätzung ist sowohl zu Beginn durchzuführen, als auch regelmäßig während des Entwicklungsprozesses um Abweichungen vorzeitig zu erkennen und Gegenmaßnahmen und Korrekturen einzuleiten.

In dieser Arbeit wurde COCOMO II als algorithmische Methode zur Aufwandsschätzung schrittweise mit seinen drei Submodellen (Application Composition Model, Early Design Model und Post-Architecture Model) und deren jeweiligen Formeln präsentiert. Auch wurden die Entwicklung des Modells und die Unterschiede zu COCOMO 81 dargelegt. Zum Abschluss wurde das Modell kritisch betrachtet, seine Einsatzgebiete erläutert und sowohl kostenlose als auch kommerzielle Tools zur Erleichterung der Berechnung vorgestellt.

Literatur

[BOEH81] B.W. Boehm, *Software Engineering Economics*, Prentice Hall, Englewood Cliffs, 1981.

[BOEH00] B.W. Boehm et al., *Software cost estimation with COCOMO II*, Prentice Hall PTR, Upper Saddle River, 2000.

[KNOE91] H.D. Knöll, *Aufwandsschätzung von Software-Projekten in der Praxis*, Bibliographisches Institut & F.A. Brockhaus AG, Mannheim, 1991.

[SOMM01] I. Sommerville, *Software Engineering*, Pearson Studium, München, 2001.

[TUWI04] o.V., *Barry W. Boehm*, URL: http://cartoon.iguw.tuwien.ac.at:16080/fit/fit01/spiral/entstehung.html, Wien, 2004.

[UNIM04] o.V., *Tools*, URL: www.unibw-muenchen.de/campus/WOW/v1052/_private/Tools.ppt, München, 2004.

Metriken zur statischen Analyse objektorientierten Source-Codes

Edmund Urbani

1. Juli 2004

Zusammenfassung

Die präzise Messung von Komplexität und Qualität von Software ist ein Ziel, das in der Informatik schon lange verfolgt wird und mit der Zeit zu immer neuen Vorschlägen für entsprechende Metriken führte. Die dabei entstandenen Software-Metriken können ein wertvolles Hilfsmittel im Entwicklungsprozeß sein. Besonders praktisch und günstig ist es, wenn die entsprechenden Metriken auch noch automatisch mit geeigneten Werkzeugen direkt auf Basis vorhandenen Source-Codes errechnet werden können. Die Zahl der Metriken, die hierfür herangezogen werden können, ist auf den ersten Blick praktisch unüberschaubar. Dies betrifft vor allem die traditionellen Metriken der strukturierten Programmierung, aber auch die Zahl der OO-Metriken ist stark im Steigen. Diese Arbeit soll einen Einstieg in den Bereich der Metriken bieten. Zunächst werden die wichtigsten "Klassiker" unter den Metriken vorgestellt und auf ihre Anwendbarkeit auf heutige objektorientierte Software hinterfragt. Der Schwerpunkt aber liegt auf den neuen OO-Metriken, die in den letzten Jahren vorgeschlagen wurden. Hier sind vor allem die Metriken von Chidamber und Kemerer von Interesse, weil diese die Grundlagen für viele weitere Entwicklungen auf diesem Gebiet schufen. Im vorletzten Kapitel werden für den praktischen Einsatz von Metriken einige freie Tools für die Programmiersprache Java vergestellt. Den Abschluß bildet eine Auseinandersetzug mit zwei wesentlichen Kritikpunkten an Software-Metriken.

1 Einleitung

Die Grundlagen für das Gebiet der Software-Metriken[1] wurden in den 60er und 70er Jahren gelegt. In den Naturwissenschaften hatte die Messtechnik bereits eine lange Tradition und war zu einem unverzichtbaren Hilfsmittel für die Forschung geworden. Man erhoffte sich mit Messungen auch in der Informatik neue Erkenntnisse zu gewinnen und Werkzeuge für die Softwareentwicklung zu schaffen. Seit jenen Anfängen sind dazu unzählige Metriken vorgeschlagen worden. Diese sollen uns unter anderem dabei helfen bessere Aufwandsabschätzungen zu treffen, den Projektfortschritt zu messen, Komplexität richtig einzuschätzen und nicht zuletzt Qualität zu messen und zu verbessern[6]. Mit der Durchsetzung des objektorientierten Paradigmas in den letzten Jahren entstanden auch neue Metriken, die die Objektorientierung entsprechend berücksichtigten. Vor allem Chidamber und Kemerer leisteten bei den OO-Metriken Pionierarbeit.

[1] In englischsprachiger Literatur wird neben dem Begriff "software metric" auch gelegentlich "software measurement" synonym dazu verwendet

1.1 Source-Code als Basis der Berechnung

Metriken in der Softwareentwicklung befassen sich mit sehr unterschiedlichen Aspekten. Das Spektrum reicht von Metriken die den Entwicklungsprozeß selbst betrachten, über Metriken für das Design bis hin zu Metriken die Aussagen über das Laufzeitverhalten (Performance, Resourcenbedarf) der entwickelten Software treffen. Diese Arbeit beschränkt sich jedoch auf Metriken, die sich auf Basis des Source-Codes einer Software errechnen lassen. Mit ihrer Hilfe sollen Aussagen über die Komplexität und Qualität des Source-Codes getroffen werden können. Wesentlich ist auch, dass diese Metriken voll automatisch - und somit kostengünstig - errechnen lassen. Sie sollen einen Einblick in die entwickelte Software geben, die Entwickler auf mögliche Probleme hinweisen und letztendlich beim Treffen von Entscheidungen im Entwicklungsprozeß helfen. Gerade bei Entwicklungsprozessen,

in denen sehr früh Code geschrieben wird[2], bieten sich solche Metriken bzw. Werkzeuge, die diese für die verwendete Programmiersprache berechnen, an. OO-Metriken sind weitgehend programmiersprachen-neutral definiert. Damit sollen sie universell auf alle objektorientierten Programmiersprachen anwendbar sein, können aber nicht immer auf alle spezifischen Merkmale einer konkreten Programmiersprache Rücksicht nehmen. Verdeutlicht wird dies in einem späteren Kapitel, daß sich mit Werkzeugen die Metriken für Source-Code in der Programmiersprache Java berechnen, befaßt.

2 Klassische Metriken

Bevor wir uns mit den OO-Metriken befassen, sollen hier klassische Metriken aus Zeiten der strukturierten Programmierung vorgestellt werden. Diese Metriken sind zwar ursprünglich nicht für objektorientierte Software ausgelegt worden, aber das bedeutet noch nicht zwangsläufig, daß sie deswegen nicht mehr anwendbar und daher uninteressant wären.

2.1 Lines of Code

Die Metrik LOC ist wohl die älteste Software-Metrik. Es gibt sie in unterschiedlichen Varianten. Die einfachste ist alle Zeilen im Code gleich zu zählen, egal ob es sich um Kommentarzeilen, Leerzeilen oder Zeilen mit Programmcode handelt. Diese Art der Zählung wird allerdings eher nicht verwendet, wenn die Komplexität und Programmgröße gemessen werden soll und Kommentarzeilen hierfür nicht relevant sind[3]. Mißt man nur Zeilen die auch Code enthalten, so spricht man von SLOC (Source Lines of Code) [7] oder eLOC (Effective Lines of Code) [8]. SLOC ist meistens gemeint, wenn man von Lines of Code spricht. Eine weitere Variante von LOC ist die Metrik LLOC (Logical Lines of Code), die nur Zeilen mit Statements zählt. Dabei wird gern genutzt, dass viele Programmiersprachen ihre Statements mit Strichpunkten beenden. Also werden einfach nur diese Zeilen gezählt, in denen zumindest ein Strichpunkt vorkommt. Zeilen, die beispielsweise lediglich eine if-Bedingung enthalten, werden dadurch allerdings nicht

[2] Dies ist zum Beispiel beim, in letzter Zeit immer mehr in Mode kommenden, eXtreme Programming der Fall.

[3] Wenn man LOC zur Messung der Leistung des Programmierers heranziehen will - wovon der Autor im übrigen abrät - sollte man die Kommentarzeilen vielleicht doch mitwerten.

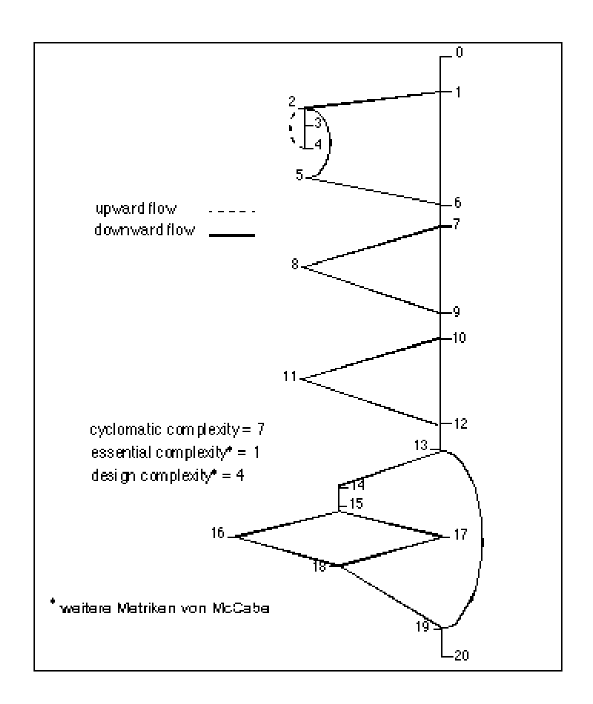


Abbildung 1: Kontrollflußgraph eines einfachen Programms[10]

gewertet.[8]
LOC ist in allen seinen Varianten sowohl auf prozedurale wie auch auf objektorientierte Programmiersprachen gleich gut bzw. schlecht anwendbar. Es ist allgemein bekannt, daß die Aussagekraft dieser Metrik eher gering ist, aber aufgrund ihrer Einfachheit, die eine sehr simple Implementierung erlaubt und die sie praktisch für jedermann - auch für Nicht-Informatiker - durchschaubar macht, ist diese Metrik bis heute nicht von der Bildfläche verschwunden. Andere Metriken werden oft mit LOC verglichen, weil als eine Grundanforderung für neuere, komplexere Metriken gilt, daß diese zumindest besser als das simple LOC sein müssen.

2.2 Cyclomatic Complexity

Tom McCabe's Cyclomatic Complexity [1][4] gilt als eine der grundlegenden Metriken der strukturierten Programmierung. Viele spätere Metriken zur Komplexitätmessung setzen auf der Arbeit von McCabe auf. Cyclomatic Complexity wird auf Basis des Kontrollflußgraphen 2.2 eines Moduls errechnet und gibt die Zahl der linear unabhängigen Ausführungspfade an.
Ist E die Anzahl der Kanten (edges) und N die

[4] In englischer Literatur auch bekannt als "program complexity" oder als "McCabe's Complexity", in deutscher Literatur auch einfach "McCabe's zyklomatische Zahl"

Anzahl der Knoten (nodes) eines Graphen G, dann errechnet man die Cyclomatic Complexity CC mit der Formel[1]:
$CC(G) = E - N + 1$
Dieser ganzzahlige Wert ermöglicht es Programmkomplexität zu vergleichen (im Allgemeinen besser als mit LOC) und läßt auf Verständlichkeit und Testbarkeit des Programmes schließen [10]. Bei objektorientierten Programmen läßt sich CC auf die einzelnen Methoden anwenden. Wie das konkret funktionieren kann, wird in Kapitel 3.2 erklärt. CC ist auch in einigen der später beschriebenen Werkzeugen implementiert[15] [16] [18].

2.3 Halstead Complexity Measures

Einen anderen Zugang zur Komplexitätsmessung fand Maurice Halstead. Er untersucht mit seinen Metriken die Komplexität auf Basis der Gesamtzahl der Operatoren N_1 und Operanden N_2 sowie der Zahl der unterschiedlichen Operatoren n_1 und Operanden n_2 in einem Modul[5]. Aus diesen Werten berechnet er die Programmlänge $N = N_1 + N_2$, das Vokabular $n = n_1 + n_2$, das Volumen $V = N * (LOG_2 n)$, die Schwierigkeit (Difficulty) $D = n_1/2 * (N_2/n_2)$ und schließlich den Aufwand (Effort) $E = D * V$.
Bei Halstead's Metriken gehen die Meinungen über deren Nützlichkeit weit auseinander [11]. Praktische Relevanz erlangten diese Metriken im Bereich der Softwarewartung als Teil der "Maintainability Index Technique for Measuring Program Maintainability"[11][12].

3 Objektorientierte Metriken

Auch wenn sich die traditionellen Softwaremetriken auf objektorientierte Software anwenden lassen, so nehmen sie doch in keiner Weise Rücksicht auf die spezifischen Merkmale der Objektoriertierung. Aus diesem Grund wurden und werden viele neue OO-Metriken entworfen, die dem objektorientierten Paradigma besser gerecht werden.

3.1 Kohäsion

Kohäsions-Metriken versuchen Aufschluß über die Zusammengehörigkeit der, zu einer Klassen zusammengefaßten Attribute und Methoden zu geben. Diese Metriken analysieren jede Klasse einzeln. Bei der Messung der Kohäsion gibt es grundsätzlich zwei verschiedene Zugangsweisen: Man versucht entweder die Kohäsion direkt zu messen oder einen möglichen Mangel an Kohäsion (Lack of Cohesion) festzustellen.

3.1.1 Lack of Cohesion in Methods

Chidamber und Kemerer leisteten mit vielen ihrer Metriken wesentliche Vorarbeit im Bereich der OO-Metriken. Unter anderem auch mit *Lack of Cohesion in Methods* (LCOM) - einer Metrik, die dazu dient, mangelnde Kohäsion festzustellen. Dazu wird in einer Klasse C mit n Methoden (M_1, M_2 bis M_n), zu jeder dieser Methoden die Menge der verwendeten Instanzvariablen I_i bestimmt. Mit diesen Mengen I_1 bis I_n werden in jeder Kombination Schnittmengen gebildet und dabei gezählt wie oft diese Schnittmenge leer ist und wie oft nicht:
$P = \{(I_i, I_j) | I_i \cap I_j = \emptyset\}$
$Q = \{(I_i, I_j) | I_i \cap I_j \neq \emptyset\}$
Wenn $|P| > |Q|$, dann $LCOM = |P| - |Q|$ sonst $LCOM = 0$.
Je öfter es keine Überschneidung gibt, desto höher ist die Zahl der Methoden zwischen denen über die verwendeten Instanzvariablen kein direkter Zusammenhang besteht, und desto höher ist somit der Mangel an Kohäsion LCOM in der Klasse [2].
Andere Autoren entwickelten neue Varianten und Definitionen von LCOM [3]. So haben etwa Hitz und Montazeri eine graphentheoretische Definition von LCOM erarbeitet, die allerdings auf einer etwas anderen Interpretation dieser Metrik von Li und Henry basiert. Später haben Hitz und Montazeri eine neue Variante von LCOM vorgestellt, die auch Methodenaufrufe berücksichtigt. In Henderson-Sellers LCOM-Metrik werden Mengen von Methoden gebildet, die auf gemeinsame Variablen zugreifen (umgekehrt wie in der ursprünglichen LCOM-Metrik von Chidamber und Kemerer). Aus dieser Variante wiederum wurde eine neue Metrik von Briand et al. entwickelt, die auch geerbte Methoden und Variablen in die Berechnung aufnimmt.
Die vorgeschlagenen LCOM-Varianten werden von ihren Autoren natürlich plausibel erläutert und ihre Vorteile gegenüber zumindest einer anderen LCOM-Metriken aufgezeigt. Aber aufgrund der Plausibilität alleine ist ein objektiver Vergleich bzgl. ihrer Praxistauglichkeit nicht möglich. Auf dieses Problem der Validierung von Metriken wird in Kapitel 5.1 noch näher eingegangen.

[5]Operatoren sind nicht nur mathematische Operatoren, sondern je nach Programmiersprache auch andere Symbole wie Indizes "[..]" oder Trennzeichen wie ";". [14]

3.1.2 Loose Class Cohesion, Tight Class Cohesion

Einen anderen Weg bei der Messung von Kohäsion verfolgen Bieman und King. Zunächst unterscheiden sie zwischen direktem und indirektem Zugriff auf Variablen. Ein direkter Zugriff ist gegeben, wenn in einer Methode unmittelbar eine Variable ausgelesen oder beschrieben wird. Bei einem indirekten Zugriff wird lediglich eine andere Methode aufgerufen, die auf die Variable direkt oder indirekt zugreift. Aus diesen direkten und indirekten Zugriffen werden die Metriken *Tight Class Cohesion* (TCC) und *Loose Class Cohesion* (LCC) berechnet. TCC ist der Prozentsatz an Methodenpaare[6], die direkt oder indirekt auf zumindest ein gemeinsames Attribut zugreifen. Es ist hier auch von direkten Verbindungen[7] zwischen den Methoden die Rede. Bei LCC werden auch indirekte (transitive) Verbindungen[8] berücksichtigt.

3.2 Komplexität

Einer Anwendung der Cyclomatic Complexity Metrik auf Objekt steht im Grunde nichts entgegen. Folgende zwei Varianten des Einsatzes in der OO-Welt bieten sich an.

3.2.1 Weighted Methods per Class

Berechnet man McCabe's Cyclomatic Complexity auf Methoden, so kann man daraus die sogenannte *Weighted Methods per Class* (WMC) von Chidamber und Kemerer bestimmen. WMC ist nichts anderes als die Summe aller CC-Werte der Methoden einer Klasse. WMC wirkt sich auf den Entwicklungsaufwand der Klasse aus. Eine Klasse mit höherer Methodenanzahl wirkt sich in der Regel auch stärker auf die möglichen Subklassen aus, weil diese auch entsprechend mehr Methoden erben. Eine Methodenzahl ist auch ein Indiz dafür, daß es sich um eine sehr anwendungsspezifische Klasse handeln könnte, was die Wiederverwertbarkeit negativ beeinflußen würde. Wie hoch der WMC-Wert einer Klasse im Optimalfall sein soll, ist nicht von vornherein klar. Software-Unternehmen können WMC verwenden indem sie Grenzwerte auf Basis der eigenen Erfahrung oder ihnen bekannten Studien festsetzen. Aber auch ohne empirische Vergleichswerte läßt sich WMC sinnvoll nutzen, um die komplexesten Klassen in einer Software zu identifizieren.

3.2.2 Average Method Complexity

Die *Average Method Complexity* von Morris [9] ist, wie der Name bereits vermuten läßt, das arithmetische Mittel der Cyclomatic Complexity aller Methoden. Eine hohe Komplexität der Methoden wirkt sich negativ auf deren Verständlichkeit aus. Dies führt zu schlechterer Wartbarkeit und höherer Fehleranfälligkeit. Daher sollte man im Allgemeinen eine eher niedrige durchschnittliche Komplexität anstreben. Allerdings sollte man sich keinesfalls alleine auf diese Metrik stützen, da man sonst Gefahr läuft eine unüberschaubar große Zahl sehr kleiner Methoden zu schreiben.

3.3 Vererbung

Wenn man objektorientierten Quellcode klassenübergreifend analysiert, ergeben sich neue Möglichkeiten für Metriken. So finden sich etwa Metriken, die sich mit der Vererbungshierarchie der betrachteten Software befassen.

3.3.1 Depth of Inheritance Tree

Chidamber und Kemerer definierten die OO-Metrik *Depth of Inheritance Tree* (DIT) [2], als die maximale Entfernung von der Wurzel des Vererbungsbaumes bis zur betrachteten Klasse[9]. Die Tiefe einer Klasse in der Vererbungshierarchie hat wesentliche Auswirkungen. Je tiefer sie sich in der Hierarchie befindet, desto höher ist potentiell die Anzahl an ererbten Methoden und Attributen. Dies führt zu einer höheren Komplexität, die es erschwert das Verhalten der Klasse korrekt vorherzusagen. Eine tiefe Hierarchie bringt eine höhere Zahl an Klassen und Methoden mit sich, als eine weniger tiefe. Aber tiefe Vererbungshierarchien ermöglichen dafür potentiell mehr Reuse. Es ist also nicht von vornherein klar, welche Tiefe des Vererbungsbaums optimal ist.

3.3.2 Number of Children

Number of Children (NOC) ist eine weitere Metrik von Chidamber und Kemerer [2]. NOC ist die

[6] Es müssen alle möglichen Paar-Kombinationen gebildet werden

[7] NDC = number of direct connections between public methods [3]

[8] NIC = number of direct or indirect connections between public methods [3]

[9] Bei Mehrfachvererbung sind mehrere unterschiedliche Distanzen zur Wurzel möglich.

Anzahl der direkten Subklassen einer Klasse. Mehr Subklassen bedeuten mehr Reuse, weil Vererbung von Methoden ja auch eine Form des Reuse darstellt. Ein sehr hoher NOC-Wert kann aber auch ein Indiz dafür sein, daß die Klasse unpassend stark abstrahiert oder das Konzept der Vererbung falsch eingesetzt wird. Generell kann davon ausgegangen werden, daß eine Klasse mit hohem NOC auch großen Einfluß auf das Design der Software hat und das diese Klasse auch ausführlich getesten werden sollte. Auch bei dieser Metrik ist es schwer allgemein gültige Aussagen über optimale Werte zu treffen. Sie kann natürlich eingesetzt werden, um Klassen mit außerordentlich hohem NOC zu finden.

3.4 Kopplung

Bei der klassenübergreifenden Analyse objektorientierter Software kann man auch Metriken einsetzen, die Aufschluß über die Abhängigkeiten zwischen den Klassen geben. Unter Kopplung fallen üblicherweise alle Abhängigkeiten außer der Vererbungsbeziehung. D.h. Aufrufe von Methoden, aber auch direkter Zugriff auf Attribute anderer Klassen, bedingen Kopplungsbeziehungen.

3.4.1 Coupling Between Object Classes

Auch zur Bestimmung der Kopplung haben Chidamber und Kemerer eine Metrik entwickelt. *Coupling between object classes* (CBO) wird für einzelne Klassen errechnet und drückt aus mit wievielen anderen Klassen diese Klasse in einer Kopplungsbeziehung steht. Die Kopplungsbeziehung der CBO-Metrik ist symmetrisch.

Hohe Kopplung ist ein Indiz für schlechtes Design. Die Klasse wird durch zuviele Abhängigkeiten untauglich für Reuse in anderen Applikationen. Abhängigkeiten zwischen Klassen sollten möglichst minimiert werden, um Flexibilität zu erhalten und so lokale Änderungen in einzelnen Klassen ermöglichen, die nicht auch gleich Änderungen in anderen Klassen nach sich ziehen. Ein hoher Kopplungsgrad weißt auch auf hohe Komplexität hin. Dementsprechend steigen Fehleranfälligkeit und Testaufwand.

3.4.2 Martins OO Design Quality Metric

Robert Martin betrachtet Abhängigkeiten zwischen Klassen etwas anders. Er versucht zwischen verschiedenen Arten der Kopplung zu differenzieren und entwickelte daher eine Metrik, die "gute" (oder zumindest nicht-schädliche) und "schlechte" Kopplung unterscheidet. Die zwischen Klassen notwendige Kommunikation führt praktisch unvermeidlich zu einem gewissen Maß an Kopplung. Martin führt des Fehlen von "schlechter" Kopplung auf gutes Design zurück. Seine *OO Design Quality* Metriken sollen auf mögliche Designprobleme, die sich in den Abhängigkeiten der Klassen zeigen, hinweisen[10][4]. Abhängigkeiten sind dann ein Problem, wenn sie auf Klassen verweisen[11], die instabil sind. Instabilität bedeutet, daß die Wahrscheinlichkeit, daß an der Klasse Änderungen durchgeführt werden, sehr hoch ist. Für Stabilität konnte Martin zwei wesentliche Indikatoren identifizieren:

Stabile Klassen haben keine Abhängigkeiten zu instabilen Klassen. Klassen, die überhaupt keine Abhängigkeiten haben, bezeichnet Martin als "Independent". Klassen, von denen viele andere Klassen abhängen, können nur schwer geändert werden. Daher gelten sie auch eher als stabil. Solche Klassen nennt Martin "Responsible". Am stabilsten sind nach seiner Auffassung Klassen, die beides - Independent und Responsible - sind.

Ein weiterer wichtiger Aspekt für Martins Metriken ist die Bildung von Klassen-Gruppen oder Packages[12]. In Packages sind Klassen mit hoher Kohäsion zueinander gruppiert. Diese Kohäsion läßt sich am Grad der Kopplung erkennen. Änderungen einer Klasse in einem Package, dürfen zu Änderungen in anderen Klassen des gleichen Package führen, aber nicht zu Veränderungen in anderen Packages. Klassen die ein Package bilden, sollen zusammen reused werden. Martin transferiert die Idee der Kohäsion von der Ebene einzelner Klassen (Kapitel 3.1) auf die Package-Ebene. Weiters argumentiert er, daß nun eigentlich die Abhängigkeiten zwischen den Packages entscheidend sind und überträgt auch seine Definition von Stabilität, "Independent" und "Responsible" auf Packages.

Schließlich definiert Martin vier Metriken:

- Afferent Couplings (Ca): Anzahl der Klassen außerhalb des Packages, die abhängig von

[10] Sie sind theoretisch, wie die meisten OO-Metriken, auch auf ein ausreichend detailliertes Design anwendbar.

[11] Im Gegensatz zu CBO ist die hier verwendete Kopplung nicht symmetrisch

[12] Martin verwendet eigentlich in Anlehnung an Booch den Begriff "Class Category". "Package" drückt das gleiche aus und ist heute nach Ansicht des Autors geläufiger.

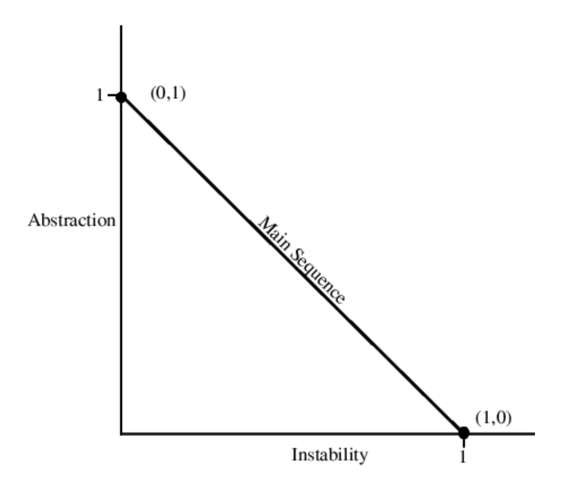


Abbildung 2: Martin's "Main Sequence"

Klassen innerhalb des Packages sind.

- Efferent Coulpings (Ce): Anzahl der Klassen im Package, die von Klassen außerhalb des Packages abhängen.
- Instability (I): $I = Ce/(Ca + Ce)$, wobei $I = 0$ maximale Stabilität und $I = 1$ maximale Instabilität bedeutet.
- Abstractness (A): Die Abstraktheit ist der Anteil abstrakter Klassen am Package ($A = 0$ bedeutet keine abstrakte Klassen, $A = 1$ ausschließlich abstrakte Klassen).

Martin setzt Abstraktheit und Instabilität in Beziehung. Packages mit hoher Stabilität sollen auch abstrakt sein, damit sie ideal erweiterbar sind. Im Gegensatz dazu sollten instabile Packages möglichst konkret sein. Packages, die auf der Linie zwischen diesen beiden Extremen liegen, der sogenannten "Main Sequence" (siehe Abb. 2), entsprechen Martin's Idealbild. Letztendlich definiert er noch die Metrik Distance $D = |(A+I-1)/2|$, die die Entfernung von der "Main Sequence" angibt. Im besten Fall ist $D = 0$, im schlechtesten Fall ist $D = 0.707$.
Martins Metriken mögen zwar einige gute Ansätze enthalten und durchwegs schlüssig erscheinen, aber trotzdem sind sie mit Vorsicht zu genießen. Tatsächlich konnte die OO Design Quality Metrik bis dato nicht validiert werden. Mehr zum Thema der Validierung kommt später in Kapitel 5.1.

4 Tools für Java

Alle bisher beschriebenen Metriken sind unabhängig von konkreten Programmiersprachen definiert. Wie einige der Metriken für die Programmiersprache Java umgesetzt wurden, wird in diesem Kapitel anhand von Open Source Tools gezeigt.

4.1 JMetric [15]

JMetric ist das Ergebnis von Forschungen der School of Information Technology an der Swinburne University of Technology. Das Programm ist frei verfügbar unter den Bedingungen der GPL[13]. In JMetric sind einige traditionelle Metriken implementiert: Statements und Lines of Code sowie Cyclomatic Complexity. Letztere wird nur auf einzelne Methoden angewandt. *Weighted Method Complexity* oder *Average Method Complexity* werden nicht berechnet.
Als einzige OO-Metriken stehen drei verschiedene Varianten von LCOM zur Auswahl - unter anderem die ursprüngliche Variante von Chidamber und Kemerer.
Das Projekt scheint zum letzten Mal im Jahr 2000 eine neue Version veröffentlicht zu haben. Der praktische Nutzen dieses Werkzeugs ist leider nicht besonders groß, vor allem weil die Dokumentation eher spärlich und die Bedienung recht mühsam ist und außerdem auch mit keiner Weiterentwicklung zu rechnen ist.

4.2 JavaNCSS [16]

Ein weiteres freies Programm zur Softwaremessung unter Java ist JavaNCSS. Leider bietet es keine echten OO-Metriken, sondern ausschließlich traditionelle Metriken. Non Commenting Source Statements (NCSS) ist eine Variante eines Statement Counters, der in diesem Programm implementiert ist. Die Cyclomatic Complexity von McCabe wird auch angeboten. Ansonsten werden nur ein paar einfache Zählungen durchgeführt (Anzahl Packages, Klassen, Methoden etc.). Metriken können global, auf Klassen-, oder auf Methoden-Ebene eingesetzt werden. Nenneswert ist die Integration in die Java Build-Tools "Ant" und "Maven" [19], die das automatische Berechnen der Metriken beim Compilieren ermöglicht.

4.3 JDepend [17]

In JDepend kommt die *OO Design Quality* Metrik von Robert Martin zum Einsatz. Bei

[13]GNU General Public License für freie Open-Source Software

6

der Umsetzung in ein Metrik-Tool, liegt es manchmal im Ermessen des Entwicklers, wie er die Metrik an die Gegebenheiten einer konkreten Programmiersprache anpaßt. Die Sprache Java ermöglicht die Definition von Interfaces. Interfaces erlauben im Gegensatz zu normalen Klassen Mehrfachvererbung, dürfen aber keinen Code beinhalten. Bei der Implementierung in JDepend hat sich Mike Clark, der Programmierer dieses Werkzeugs, entschieden, Interfaces auch einfach als abstrakte Klassen zu zählen.
Mit JDepend ist sowohl die interaktive Analyse von Klassen und Packages mit der enthaltenen GUI, als auch der Batch-Betrieb zur automatisierten Erzeugung von Reports möglich. Clark beschreibt sogar wie man JDepend mit JUnit kombinieren kann, um den Build-Prozeß scheitern zu lassen, wenn sich die Werte der Metrik außerhalb der festgesetzten Schranken befinden. Auch die Integration mit Ant und Maven ist vorgesehen.
Die Reports können mit steigender Klassenanzahl schnell recht umfangreich werden, da wirklich alle von Martin vorgeschlagenen Kennzahlen errechnet werden. Die Dokumentation von JDepend ist sehr zufriedenstellend und es scheint immer noch weiterentwickelt zu werden.

4.4 CCCC [18]

Diese Metrik-Software wurde eigentlich fur C/C++ geschrieben, aber inzwischen unterstützt sie auch Java. CCCC bietet zahlreiche klassische Metriken (LOC, CC, Fan-in Fan-out). *Weighted Methods per Class* (WMC) ist die einzige bisher implementierte OO-Metrik. Auch CCCC ist freie Open-Source Software.

4.5 Fazit

Das Angebot freier Software ist etwas enttäuschend. Für OO-Metriken finden sich kaum Werkzeuge, traditionelle Metriken sind noch eher vertreten. Anscheinend ist in der Open Source Community kaum jemand vom Nutzen der OO-Metriken überzeugt. Dies mag wohl auch an der Art und Weise der Softwareentwicklung und der Mentalität in der Community liegen.

5 Kritikpunkte

Auch wenn Software-Metriken das Potential haben, den Softwareentwicklern die Arbeit mit wertvollen, quantifizierten Informationen zu erleichtern, soll in dieser Arbeit nicht verschwiegen werden, dass ihr Einsatz in der Praxis nicht unumstritten ist. Zwei der Kritikpunkte an Metriken werden in den folgenden Unterkapiteln näher erläutert.

5.1 Validierung

Die Zahl der vorgeschlagenen OO-Metriken ist jetzt schon kaum überblickbar und natürlich ist davon auszugehen, dass auch weiterhin neue Metriken hinzukommen. Bei der raschen Weiterentwicklung kommt die Validierung oft zu kurz. Vor allem bei ganz neuen Ansätzen wäre ein Validierung wichtig, um zu sehen, ob der neue Weg sinnvoll ist. So konnte etwa Robert Martins "OO Design Quality" Metrik nicht validiert werden, obwohl sie zunächst durchaus plausibel erscheinen mag und einige interessante neue Ideen enthält.
Auch bei Varianten bereits bekannter Metriken, wäre es notwendig zu überprüfen, ob durch die Änderung auch tatsächlich eine Verbesserung gegeben ist. Ein Versuch einige OO-Metrik zu vergleichen wurde an der University of Alabama durchgeführt[3]: Dabei wurden mehrere Kohäsionsmetriken[14] an zwei C++ Klassenbibliotheken errechnet. Zwei Expertenteams bewerteten evenfalls diese Klassen hinsichtlich Kohäsion. Die höchste Korrelation zur Meinung der beiden Teams wurde bei der LCC-Metrik festgestellt. Bei LCOM und seinen Varianten wurde keine eindeutige Reihenfolge ersichtlich. Am schlechtesten schnitt TCC ab. Das Ergebnis fällt zwar recht eindeutig zu gunsten von LCC aus, dennoch sollte man wegen dieser Untersuchung nicht gleich davon ausgehen, dass diese Metrik eindeutig die beste Kohäsionsmetrik ist. Der gleiche Versuch mit anderen Klassenbibliotheken oder anderen Experten könnte relativ stark abweichende Ergebnisse liefern.
Ein Problem der Validierung selbst ist, das unstandardisierte, und in Folge teilweise sehr unterschiedliche Vorgehen. Bemühungen, die Vorgehensweisen zu vereinheitlichen, unternahmen u.a. Briand, Emam und Morasca[21]. Sie identifizierten 2 Arten der Validierung, die beide notwendig sind und einander ergänzen sollen:

a) Theoretische Validierung
Bei dieser Art der Validierung wird überprüft, inwieweit die Metrik tatsächlich mit der

[14] U.a. alle in Kapitel 3.1 erwähnten

Eigenschaft korreliert, die sie laut ihrem Erfinder messen soll.

b) Empirische Validierung
Diese Validierung konzentriert sich auf den Nutzen der Metrik in der Praxis. In die empirische Validierung fließt daher mit ein, wie die Metrik tatsächlich eingesetzt wird.

Auch von anderer Seite wie zB. von Kitchenham et al wird versucht die Validierung von Metriken zu standardisieren. Von ihr wird ein "Framework for Software Measurement Validation"[22] vorgeschlagen.
Die Entwicklungen im Bereich der Metriken-Validierung sind vielversprechend und geben Grund zur Hoffnung, dass die bereits vorhandenen und auch neue Metriken möglichst objektiven Überprüfungen unterzogen werden. Außerdem sollten sie es Software-Firmen ermöglichen selbst Metriken im Rahmen eines Projektes einer klar definierten Validierung zu unterziehen.

5.2 Druckmittel Metrik

Der zweite Kritikpunkt betrifft weniger die Metriken an sich, als vielmehr die Art und Weise wie sie verwendet werden bzw verwendet werden könnten. Mit dem Thema, wie Metriken in einem Unternehmen benutzt werden, um Druck auf die Mitarbeiter auszuüben, hat sich Tom DeMarco in seinem Artikel "Mad About Measurement"[5] ausführlich beschäftigt. Er zeigt anhand von realen Beispielen innerhalb und außerhalb der Software-Industrie, wie sich Arbeiter und Angestellte dazu veranlasst sehen können, die Metriken, an denen sie gemessen werden, zu umgehen. DeMarco nennt dies eine Dysfunktion. Diese tritt ein, sobald Mitarbeiter Energie darauf verwenden, die Metrik zu ihrem eigenen Vorteil zu beeinflussen, ohne dabei die ursprüngliche Intention mit der die Metrik eingeführt wurde (etwa Effizienzsteigerung, Qualitätssteigerung,..) zu verfolgen. So kann durch den Einsatz einer Metrik sogar die Qualität sinken. Keine der in dieser Arbeit angeführten Metriken ist darauf ausgelegt, unüberlistbar zu sein. Die Annahme der Autoren dieser Metriken ist, dass die Softwareentwickler daran interessiert sind, gute Arbeit zu liefern und nicht Metriken zu manipulieren. Selbst wenn es eine Metrik gäbe, die zB. unverfälschlich die Qualität einer Software quantifizieren könnte, wäre sie wahrscheinlich in der Praxis nicht sehr hilfreich. Denn sie müßte wohl so kompliziert sein, dass sie niemand versteht und würde keine Rückschlüsse ermöglichen, wie denn die Qualität verbessert werden könnte.

Die meisten der, in dieser Arbeit beschriebenen, Metriken sind wohl eher ungeeignet, um sie dem Management zu präsentieren. Dementsprechend gering ist bei ihnen die Gefahr, dass diese Dysfunktion eintritt. Dieses Risiko besteht vor allem bei Metriken, die einen, auch für Nicht-Informatiker verständlichen Output produzieren. Darunter fallen einerseits sehr simple Metriken wie LOC und andererseits komplexe Metriken, die versuchen ein sehr einfaches aber aussagekräftiges Ergebnis zu liefern, wie Martins "OO Design Quality" Metrik.

6 Conclusio

Wäre der Nutzen von Software-Metriken in der Praxis unumstritten, würden sie wohl schon längst viel mehr Unternehmen und auch Open-Source Entwickler einsetzen. Überraschenderweise hat anscheinend gerade die kontroversielle Metrik von Robert Martin in der Open Source Community etwas Anklang gefunden.
Trotz der Schwächen und aller Kritik die zum Thema Software-Metriken geäußert wurde und auch heute noch geäußert wird, sehe ich in ihnen vielversprechende Werkzeuge für die zukünftige Softwareentwicklung. Ob und wie man sie am besten zum Einsatz bringt hängt nicht zuletzt vom gewählten Entwicklungsprozeß ab. Gerade Methoden der agilen Softwareentwicklung wie eXtreme Programming, die schon sehr früh mit der Implementierung beginnen, könnten von Tools, die auf Basis des Source-Code Metriken zur Verfügung stellen, sehr profitieren. Einige der OO-Metriken ließen sich auch auf das Design anwenden - es wären nur andere Tools zu deren Berechnung notwendig.
Gute Verwertungsmöglichkeiten für Metriken scheinen mir auch als Hilfsmittel für Reviews gegeben. Hier könnten schnell Methoden und Klassen ausfindig gemacht werden, deren Metriken auffällig abweichen (zB. hohe WMC einer Klasse). Im Rahmen eines Reviews ließe sich der Ursache für die Auffälligkeit auf den Grund gehen und diskutieren, ob eine Änderung notwendig ist.
So lange man, OO-Metriken rein als Unterstützung für den individuellen Entwickler oder für ein Entwicklerteam betrachtet, ist auch das Risiko der von DeMarco beschriebenen Dysfunktion sehr gering. Wohl überlegt sollte eine Weitergabe von Messdaten an das Management sein. Sollte es auf Software-Metriken bestehen, empfehle ich den Daten DeMarcos Artikel[5] beizufügen.

Literatur

[1] McCabe, Thomas J., Butler, Charles W., "Design Complexity Measurement and Testing." Communications of the ACM 32, 12 (December 1989)

[2] Chidamber, Kemerer, "A Metrics Suite for Object Oriented Design", IEEE Transactions on Software Engineering 20 (1994) 476 - 493

[3] Etzkorn, Gholston, Fortune, Stein, Utley, Farrington, Cox, "A comparison of cohesion metrics for object-oriented systems", Information and Software Technology (http://www.elsevier.com/locate/infsof), December 2003

[4] Martin R., "OO Design Quality Metrics - An Analysis of Dependencies", http://www.objectmentor.com/ publications/oodmetrc.pdf, October 1994

[5] Tom DeMarco, "Why Does Software Cost So Much?", "Essay 2: Mad About Measurement", Dorset House Publishing, 1995

[6] Zuse Horst, "History of Software Measurement", http://irb.cs.tu-berlin.de/ zuse/metrics/3-hist.html

[7] Wheeler David A., "SLOCCount", http://www.dwheeler.com/sloccount/

[8] Jeff Nyman, "Code Metrics", http://www.globaltester.com/sp6/ codemetrics.html

[9] Morris, "Metrics for Object-Oriented Software Development Environments. Master's Thesis", M. I. T. Sloan School of Management, 1989

[10] VanDoren Edmond, "Cyclomatic Complexity", http://www.sei.cmu.edu/str/descriptions/ cyclomatic.html

[11] VanDoren Edmond, "Halstead Complexity Measures", http://www.sei.cmu.edu/str/descriptions/ halstead.html

[12] VanDoren Edmond, "Maintainability Index Technique for Measuring Program Maintainability", http://www.sei.cmu.edu/str/descriptions/mitmpm.html

[13] Sencer S. "Software Measurement Page, Object Oriented Metrics", http://yunus.hun.edu.tr/ sencer/oom.html

[14] Sencer S., "Complexity Metrics and Models", http://yunus.hun.edu.tr/ sencer/complexity.html

[15] Cain Andrew, "JMetric Home Page", http://www.it.swin.edu.au/projects/ jmetric/products/jmetric/

[16] "JavaNCSS", http://www.kclee.com/clemens/ java/javancss/

[17] Clark Mike, "JDepend", http://www.clarkware.com/software/ JDepend.html

[18] Littlefair Tim, "Software Metrics Investigation", http://cccc.sourceforge.net/

[19] Apache Software Foundation, "Jakarta Project", http://jakarta.apache.org/

[20] Dumke Reiner, "Softwaremetrie", http://ivs.cs.uni-magdeburg.de/ dumke/Metrie/Smetrie.html

[21] Briand, Emam, Morasca, "Theoretical and Empirical Validation of Software Product Measures",http://www.software-metrics.org/documents/isern-95-03.pdf

[22] Kitchenham, Pfleeger, Fenton, "Towards a Framework for Software Measurement Validation", IEEE Transactions on Software Engineering 21 (1995) 929-944

Management of Requirements Traceability Problems

Jörgl Kerstin
Mat. Nr. 0060260
kjoergl@edu.uni-klu.ac.at

Abstract

Requirements-Traceability ist für die Entwicklung von Software-Systemen mit hoher Qualität wesentlich. Während das Traceability, dass zur Verfeinerung, Entwicklung und zum Gebrauch von einer Anforderung, Post-Traceability genannt wird, wird das Tracing einer Anforderung zurück zu seinem Ursprung pre-traceability genannt. In dieser Arbeit wird das grundlegende Problem des Requirements Tracing (RT) analysiert, besprochen, verarbeitet und Möglichkeiten der Verbesserung präsentiert. RT wird als Prozess der Dokumentation identifiziert, da einerseits die Verbindungen zwischen Benutzeranforderungen an das System begutachtet werden und andererseits das Produkt, das entwickelt wird, um jene Anforderungen einzuführen und zu überprüfen. Zusätzlich wird in dieser Arbeit noch über die Qualitätsaspekte durch den Einsatz des Requirements-Tracing und den damit verbunden Kosten referiert.

1. Introduction

Bei RT geht es im engeren Sinn um die Fähigkeit, das „Leben" einer Anforderung während des Life Cycles zu beschreiben und zu verfolgen. In diesem Zusammenhang wäre zu notieren, dass Traceability als eine unentbehrliche Verbindung oder definierbares Verhältnis zwischen „Wesen" anzusehen ist. Die Anforderung ist eine Softwarefähigkeit, die durch ein System oder einen Bestandteil getroffen sein muss, um einen Vertrag, eine Spezifikation, einen Standard oder ein anderes formales Dokument zu erfüllen [1].

In dieser Arbeit wird das grundlegende Problem des Requirements Tracing und des Requirements Traceability analysiert. Im Kapitel 2 wird zu Beginn genau beschrieben, warum Requirements Tracing betrieben werden sollte und vorallem wie.

Im anschließenden Kapitel 3 wird auf die 2 Hauptarten des RT eingegangen: einerseits das Pre-Traceability und andererseits das Post-Traceability. Hier werden von mir die möglichen Probleme identifiziert und die möglichen Lösungsansätze und Support-Möglichkeiten kurz aufgezeigt. Weiters wird noch kurz beschrieben, welche Informationen benötigt werden und wie diese darzustellen sind.

In Sektion 4 wird danach das Traceability Model mit seinen 10 Eigenschaften analysiert und wie einzelne Mitarbeiter eines Projektes RT nutzen können. Abschließend werden noch einige Tools vorgestellt

Im Abschnitt 5 wird dann erläutert, wie Requirements Tracing zur Qualitätsverbesserung beitragen kann und abschließend wird noch auf den Traceability-Benefit und die damit verbundene Reduktion der Life-Cycle-Kosten eingegangen.

2. Why and How of Requirements Tracing

Dieses Kapitel der Arbeit widmet sich dem „Warum" und „Wie" des Requirements Tracing. Wie schon erwähnt wurde, handelt es sich bei Requirements Tracing um einen Prozess der Dokumentation, der einerseits die Verbindung zwischen Benutzeranforderungen an das System begutachtet und anderseits das Produkt an sich, das entwickelt wird, um jene Anforderungen einzuführen und zu überprüfen.

2.1 Das „Warum" des RT?

Die Autoren des Papers „Why and How of Requirements Tracing", Robert Watkins, Softwareingenieur und Mark Neal, Software-Produktmanager, entwickeln schon jahrelang Software für das US Verteidigungsministerium und sie identifizierten 4 Ziele, warum Requirements Tracing betrieben werden sollte [2]:

2.1.1 Verifikation

Verifikation hilft uns Softwareanforderungen zu überprüfen. Wenn etwas den Anschein macht, nicht der Norm zu entsprechen, können gegebenenfalls frühzeitig Verfahren entworfen werden. Diese werden wiederum koordiniert und letztendlich wiederum getestet.

Denn nur so kann gewährleistet werden, dass alle erforderlichen Funktionen, Anforderungen und Designkomponenten im Produkt vorhanden sind

2.1.2 Kostenreduktion

Traceability lässt alle Produktanforderungen sehr früh im Life-Cycle der Entwicklung zuteilen und somit erheblich Geld sparen. Die Kosten die sonst beim „Warten“ entstehen, bis Integration und System-Test Phase abgeschlossen ist, um Defekte zu beheben, ist ein fixer Bestandteil und 30 mal stärker als bei der vorgesehenen Ausgangsphase.

2.1.3 Verantwortlichkeit

Während der internen oder externen Bilanzen hat das Projekt eine bessere Erfolgsrate, wenn die Daten für Revisoren vorhanden sind und sie prüfen können, ob eine Anforderung erfolgreich durch verbundene Testfälle validiert wurde. Projektmanager haben einen besseren Handgriff bez. Kosten und dem Kunden wird beim Erhalt des Produktes versichert, dass er bekommt was er verlangt hat. Indem man quantitative Analysen zur Verfügung stellt, können Projektmeilensteine leicht genehmigt und so leichter überprüft werden, was wiederum die Kundenzufriedenheit erhöht und das Vertrauen steigert.

2.1.4 Change Management

Für jede Änderung ist es einfacher, festzustellen, in welcher Verbindung die Elemente zum Design stehen und wie diese dadurch beeinflusst werden. Es dient dazu, die laufenden Dokumente immer auf dem neuesten Stand zu halten, während die Implementierung weiterläuft. Zusätzlich können Manager Testverfahren kennzeichnen, die die Änderungen wieder verifizieren sollen. Dieses Wissen speichert Testressourcen und garantiert eine Einhaltung des Zeitplans.

2.2 Das „Wie“ des RT?

Benutzt werden Prozesse, Techniken und Werkzeuge, um Verhältnisse aufzubauen und beizubehalten und getrennte Eingänge und Ausgänge in den Entwicklungsphasen aufeinander zu beziehen.

Die unten abgebildete Graphik zeigt die Phasen der Entwicklungen und den Anfang und die schließlich entstandenen Endprodukte, die mit jedem verbunden sind.

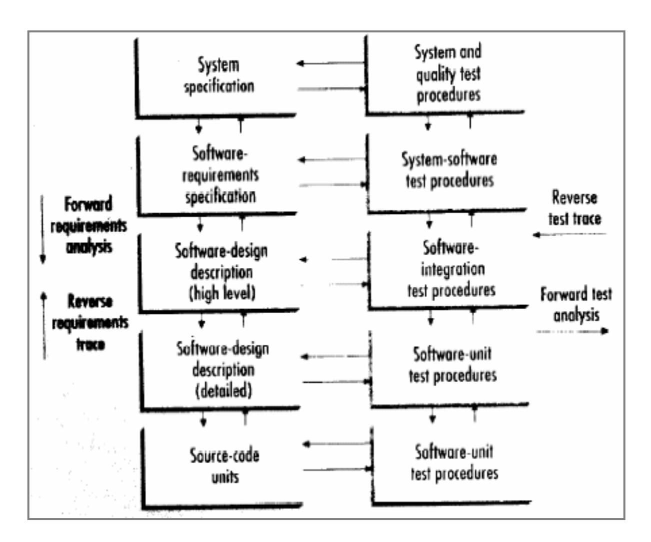


Abb.1: Ablauf „Wie“ RT passiert [2]

3.4 Welche Informationen müssen notiert werden?

Wie die Überschrift schon verdeutlicht, stellt sich die Frage, welche Arten von Informationen müssen überhaupt notiert werden? Requirements Pre-Traceability wird hergestellt, wenn der Anforderungsdefinitionsprozess selbst nachvollziehbar ist. Somit eine kurze Antwort auf die Frage „Welche Art der Informationen müssen notiert werden“. Um Pre-Traceability jedoch zu ermöglichen, müssen die Anforderungen im Rahmen eines Anforderungsprozesses definiert sein [12].

Hierfür dient ein 3-dimensionales Gerüst, das aus den Achsen

- Specification
- Representation und
- Agreement

besteht [4].

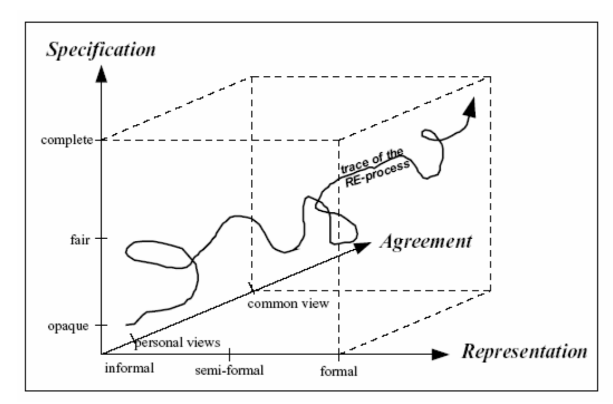


Abb. 2: Das 3-dimensionale Framework [4]

- **Dimension der Repräsentation** - als Grundlage für die formale Entwicklung, bewegt sich die Anforderungsdimension von einer anfänglich ziemlich formlosen zu einer formalen Repräsentation, die Idealerweise mit einer formalen Spezifikation endet und leicht in nachvollziehbaren Code umgewandet werden kann. Diese Dimension ist leider nicht sehr einfach zu modellieren, da unterschiedliche Personen in bestimmten Situationen eine andere Art der Darstellung bevorzugen. Wenn diese Darstellungen modelliert werden, kommt es auf dieser Linie meist zu technischen Problemen, die eine Mensch-Computer Interaktion erfordert [12].

- **Dimension der Spezifikation** - ein zweites offensichtliches Ziel ist es, mit einer kompletten Systemspezifikation abzuschließen, die jedermann versteht und nachvollziehen kann, auch ohne viel Systemverständnis. Hier sind jedoch 2 Arten der Vollständigkeit zu notieren: die erste Art beschäftigt sich mit der Abdeckung des Problems in der Spezifikation, sprich alle relevanten Anforderungen müssen hier abgefangen werden. Die zweite Art bezieht sich auf den Standart der Anforderungen, sprich alle Anforderungen müssen in dem zuvor definierten Standart spezifiziert worden sein, um so die Korrektheit des Systems zu gewährleisten [4]. Außerdem wird das gleiche Wissen häufig in den unterschiedlichen Darstellungen beschrieben, z.B. wörtlich und graphisch. Wenn man sich entlang dieser Linie bewegt, kommt es sehr oft zu kognitiven und psychologischen Problemen [12].

- **Dimension der Vereinbahrung** - Diese Achse betrifft die Vereinbahrungen, die zu Beginn getroffen wurden, und die bei der gegenwärtigen Spezifikation zu erreichen sind. Auf Grund dessen, dass das Requirements Engineering hier als ein Prozess der Vermittlung und des Lernens angesehen wird, sollte diese Vereinbarung getroffen werden, bevor begonnen wird, das System zu entwickeln. Folglich beschäftigt sich diese Linie hauptsächlich mit Sozialaspekten der Anforderungsdefinition [12].

Abschließend wäre hier zu vermerken, dass bei der Gestaltung der Informationen während des Systementwicklungsprozesses (SEP) drei grundlegende Aspekte zu beachten sind [12].

- Die Verwendung von Trace-Informationen ist stark von der jeweiligen Personengruppe und der Phase des Softwareentwicklungsprozesses abhängt. Das heißt, dass Trace-Informationen später in einem anderen Kontext verwendet werden, als sie vorher aufgezeichnet wurden. Die Strukturierung und Verwaltung der erfassten Daten muss also spezifische Sichten und ein selektives Retrieval gemäß dem aktuellen Bedarf bei der Verwendung unterstützen.
- Zweitens bietet aufgrund der großen Informationsmenge, die während der Prozessdurchführung anfällt, nur eine Inhaltsorientierte, in einen breiteren Prozesskontext eingebettete Erfassung von Trace-Informationen eine Basis zur geeigneten Wiederverwendung.
- Drittens sind die Personen, die in die Trace-Erfassung involviert sind, nicht identisch mit den Verwendern der aufgezeichneten Trace-Informationen. Daher wird eine standardisierte Traceability-Struktur benötigt, die garantiert, dass Trace-Informationen stets auf die gleiche Art erfasst werden, um dadurch eine einfachere Verwendung durch Dritte zu ermöglichen [8].“

3. Arten des Requirements Tracing

Vorweg wäre zu diesem Thema zu vermerken, dass es alles in allem 4 Arten des Requirements Tracings gibt [3]:

Forward from Requirements: Hier geht es um die Zuweisung der Verantwortlichkeit für die Ausführung der Anforderungen an die Systembestandteile. Es muss gewährleistet werden, dass die Verantwortlichkeit hergestellt wird und die Auswirkung der Anforderungsänderungen ausgewertet werden kann.

Backward to Requirements: Hierbei wird überprüft, ob das Systems die Anforderungen richtig umsetzt. Weiters muss hier das Prinzip des „Gold-plating“ (z.B.: ein Design, für das keine Anforderungen bestehen) vermieden werden.

Forward to Requirements: Bei diesem Punkt geht es um die Änderung der Bedürfnisse des Kunden. In technischen Annahmen, kann es eine radikale Neuveranlagung der Anforderungen bedeuten.

Backward from Requirements: Die zu Grunde liegenden Anforderungen der Beitragsstruktur sind entscheidend, wenn man Anforderungen, besonders in einem hohen Grade validiert.

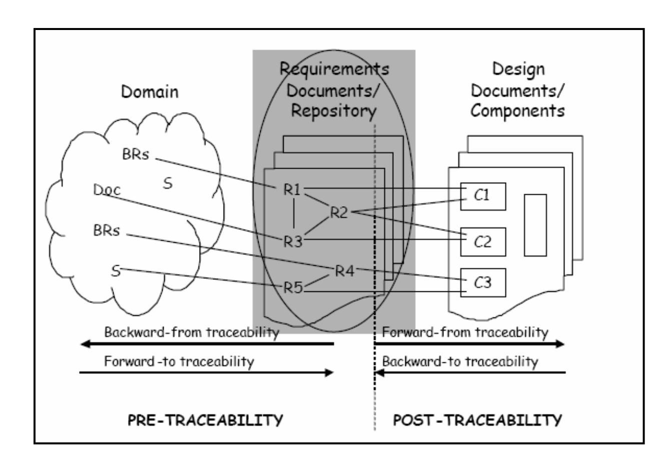


Abb. 3: Die 4 verschieden Typen des RT [13]

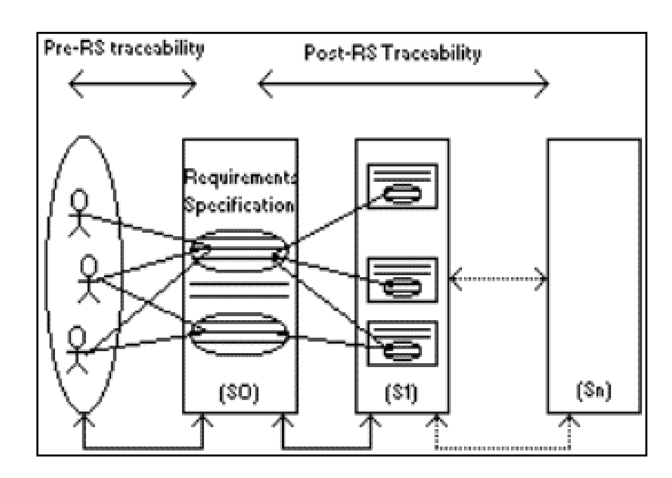


Abb. 4: 2 Typen des Requirements Tracing [1]

Einerseits dient „Traceability from the Requirements Specification“ vom Entwurf bis zur Implementierung des besseren Verständnisses des gegenwärtigen Systems. Andererseits muss der Entwicklungsprozess „To the Requirements Specification“ nachvollziehbar sein, um die Anforderungen an sich zu verstehen und sie an ihren Ursprung zurück zu verfolgen[5].

Wie bereits bekannt ist, dient Requirements Traceability zur Entwicklung von Software-Systemen mit sehr hoher Qualität. Während das Traceability der Verfeinerung, der Entwicklung und des Gebrauches von einer Anforderung „Post-Traceability“ (in der Literatur auch als Forward Traceability bekannt) genannt wird, wird das Traceability einer Anforderung zurück zu seinem Ursprung „Pre-Traceability“ (in der Literatur auch als Backward Traceability bekannt) genannt [4].

Die nächsten 2 Unterkapitel von Requirements Tracing behandeln Pre-Traceability und Post-Traceability. Bevor jedoch erklärt wird, wie diese 2 Arten des Requirements Tracing von statten gehen, gibt es vorweg noch eine kurze Erklärung: [1]

Pre-Traceability – dazu werden Aspekte des täglichen Lebens in die Requirements Specification (RS) der Produktion miteinbezogen.

Post-Traceability – hierbei fließen die Resultate, die bei der Entwicklung eines Produktes entstehen, post-mortem in die RS ein.

Wie Eingangs schon notiert wurde, ist der Ausgang bei Post- und Pre-Traceability immer von einer Requirements Specification (RS). Diese RS muss (sollte) folgende 7 Qualitätsattribute aufweisen [13]:

- **Komplettheit** - Eine Anforderungsspezifikation sollte alle relevanten und bedeutenden Anforderungen des Systems einschließen, „functional“ – dass sind jene, die sich mit den Themen Funktionen und Features beschäftigen - und „non-functional“ – das sind die, die sich auf Performance und Qualität beziehen.
- **Konsistenz** – sprich, die Übereinstimmung des Dokumentes, sollte betreffend Gebrauch von Terminologie und ähnlicher Funktionalität sichergestellt sein z.B.: der Button „Close“ muss in allen teilen des Systems gleich ausschauen.
- **Modifizierbarkeit** - Das Dokument sollte in einer zusammenhängenden Weise organisiert sein, da es so einfacher zu ändern und zu verwenden ist. Z.B. sollten Inhaltsverzeichnisse einen Hyperlink auf den Teil im Dokument beinhalten.
- **Nachvollziehbarkeit** – Sie muss zwischen ähnlichen Dokumenten garantiert werden. Das heißt, dass die Nachvollziehbarkeit rückwärts zur Quelle sichergestellt sein muss, genauso wie vorwärts zum Design und zum Code.
- **Eindeutigkeit** - Die Aussage über die Anforderungen sollte eindeutig sein, das heißt, die Anforderung darf nicht in unterschiedlichen Weisen zu deuten sein. Eine Aussage sollte nicht mehr als eine Bedeutung haben.
- **Verifizierbarkeit** - Es sollte möglich sein, die Anforderungen zu überprüfen, dass heißt die Anforderungen müssen erfüllt werden. Die Anforderungen im Dokument sollten hingegen messbar sein.
- **Verwendbarkeit während der Entwicklung, des Betriebes und der Wartung des Systems** - Das Dokument sollte in der vollständigen Lebensdauer des Informationssystems verwendbar sein und bedeuten, dass es in einer Weise ent-

worfen werden sollte, die als Grundlage für die Arbeit dienen kann, welche während der Entwicklung, der Betriebstechnik eines Informationssystems erledigt wird.

Diese Eigenschaften sind wichtig, um zu verfolgen, dass die Anforderungsspezifikation in den verschiedenen Situationen während der Entwicklungsprozesse verwendet werden kann. Die Spezifikation muss eine komplette und gleich bleibende Ansicht des Systems reflektieren, die nicht in den unterschiedlichen Weisen gedeutet werden kann.

3.1 Pre-Traceability

Wie schon notiert wurde, wird Traceability als die Fähigkeit beschrieben und definiert, um das Leben einer Anforderung in beiden Richtungen zu verfolgen. Die Pre-Traceability behandelt weiters die Aktivitäten, die vor oder/und während der RS auftreten. Hier geht es aber nur um die Rückwärtsrichtung [6].

Die Rückwärtsrichtung entspricht der „Pre-Anforderung", dass heißt, die relevanten Aspekte, die jene Anforderung nach oder während ihres „Lebens" durchläuft, werden in die Dokumentation miteinbezogen. Pre-Traceability hängt von der Fähigkeit der Nachvollziehbarkeit im Prozess der Produktion und der ständigen Verfeinerung zu ihrem Ursprung ab, nach vor sowie auch zurück. Es muss auch gewährleistet werden, dass alle Statements von verschiedenen Quellen in die vorhandene Spezifikation integriert werden. Es ist auch offensichtlich, dass die Eigenheiten der Beteiligten eine große Rolle spielen und sich im Bezug auf den Support der Anforderungen auswirkt [7].

3.1.1 Probleme des Pre-Traceability

Im Prozess des Requirements Tracings – speziell hier, beim Pre-Traceability – gibt es eigentlich nur 2 große Gruppen der Teilnehmer, die Probleme haben und ihren Bedarf irgendwie zum Ausdruck bringen wollen: erstens die SW-Entwickler und zweitens die End-User. Die SW- Entwickler benötigen Hilfe bei Verbindung des Pre-Traceability mit ihrer Arbeit. Auf Grund dessen, muss der SW-Entwickler dem Pre-Traceability eine höhere Priorität und somit mehr Zeit zuweisen, als vergleichsweise der erheblichen Systemverbesserung[10].

Die Endbenutzer benutzen Pre-Traceability, um ihre Bedürfnisse zum Ausdruck zu bringen. Die nun angeführten Probleme und die dazugehörige Graphik soll dies veranschaulichen [1]:

- Ein stereotypischer Endbenutzer kann nicht vorbestimmt werden. Seine Anforderungen unterscheiden sich und sind häufig inkonsistent.
- Die Quantität, die Uneinheitlichkeit und die Tiefe des Details der benötigten Informationen, schließt Pre-Definitionen aus.
- Auf Grund der Unfähigkeit, vorzubestimmen, wie jeder Zugang zu den Informationen bekommt und wie sie darzustellen sind, muss dies gefordert und zu Beginn – in der Spezifikation - festgelegt werden.
- Vertrauen auf persönlichen Kontakt und Korrektheit, da es immer irgendwelche Daten gibt die, veraltert, unbrauchbar, undokumentiert und unzugänglich sind.
- Jeder Endverbrauchkontext stellt einzigartige Anforderungen aus, also entstehen Probleme, wenn Endbenutzer nicht die Fähigkeit haben, die Informationen zu filtern und zugänglich zu machen. Diese Probleme betreffen die RS Produktion, die Endbenutzer unter den unterschiedlichsten Umständen benötigen.

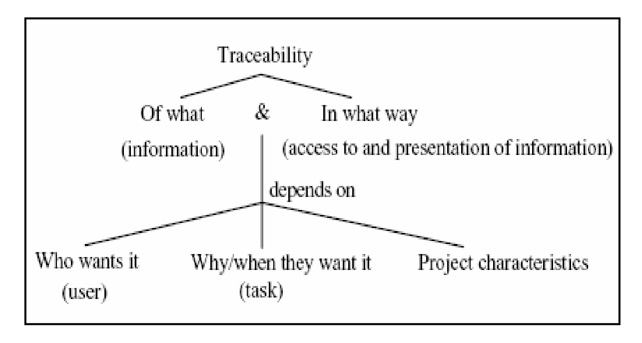


Abb. 5: Probleme des Endbenutzers [1]

3.1.2 Lösungen bzw. Lösungsansätze

Ein RS wurde produziert, um zu spezifizieren, was angefordert wird und um Pre-Traceability zur Verfügung zu stellen und zu gebrauchen. Die Kompliziertheit dieser Anforderungen zeigt an, dass es voreilig sein würde, eine komplette Lösung anzubieten. Es benötigt ein zusammengesetztes Problem, dass zu Verbesserungen dient und in vielen verschiedenen Bereichen einsetzbar ist. Anbei die 4 wichtigsten [1]:

- **Zunehmender Fluss an Informationen** – jeder sollte nur die Informationen erhalten, die er auch wirklich benötigt und nicht generell alle.
- **Erreichen und Notieren von Informationen** – mit welchen Tools kann ich an die Informationen kommen.

- **Organisieren und Beibehalten von Informationen** - jeder sollte wissen, wer was macht und von wem ich meine eventuell benötigten Informationen bekomme.
- **Zugang und Darstellung der Informationen** – wenn jemand Zugang zu den Daten benötigt, sollte ihm dieser unter Sicherheitsaspekten gewährleistet werden. Weiters muss die Darstellung der Daten vereinheitlicht werden.

Diese Punkte wurden aber schon ausführlich in Kapitel 3.4 behandelt.

3.2 Post-Traceability

Bei Post-Traceability handelt es sich um die Nachvollziehbarkeit der Anforderungen vorwärts – hindurch durch die Verfeinerung, die Entwicklung und die Verwendung eines Systems - um die neueren Stadien zu begutachten und sie von jenen neuen Artefakten wieder zurück, auf die ursprüngliche Spezifikation zu verfolgen. Die Anforderungsspezifikation fungiert als eine Art Grundlinie, von der aus verglichen wird. Verglichen werden die Anforderungen, die bis jetzt umgesetzt wurden, mit jenen, die zu Beginn spezifiziert wurden. Wenn Änderungen an den Anforderungen vorgenommen werden, müssen sie durch alle Artefakte des Systems fortgepflanzt werden, die sich auf jene geänderten Anforderungen beziehen [10].

3.2.1 Probleme des Post-Traceability

Das Post-Traceability bezieht sich auf die Aspekte der Anforderungen zu jenem Zeitpunkt, wenn sie in die Anforderungsspezifikation mit eingeschlossen werden. Nachdem dies nicht gerade ein leichtes Unterfangen ist, kommt es auf Grund der Fülle der Anforderungen öfters zu Problemen. Es werden entweder Anforderungen verloren, nicht mit übernommen und/oder welche hinzuaddiert. Requirements Post-Traceability ist auch für die Änderung der Integrationen wichtig, da sie eine Kennzeichnung der Entwicklung von Design- und Implementierungsänderungen ermöglicht.

3.3 Support von Pre- und Post-Traceability

Vorhandene Unterstützung liefert hauptsächlich Post-Traceability. Alle möglichen Probleme sind ein Artefakt der formlosen Entwicklung von Methoden. Diese können durch formale Entwicklungseinstellungen beseitigt werden, wenn sie automatisch in eine ausführbare RS transformiert werden könnten – sie ist aber ohnehin nur schwer verwendbar. Demgegenüber steht die Aussage, dass Pre-Traceability nur schwer verständlich ist und die Nachvollziehbarkeit nicht bzw. nur schlecht unterstützt. Es ist argumentiert worden, dass Probleme des Pre-Traceability, ungeachtet der formalen Behandlung bleiben, da dieser Aspekt des Lebens einer Anforderung in sich selbst Paradigma unabhängig ist [1].

4. Requirements Traceability Meta-Model

Nachdem nun ein kurzer Ein- und Überblick über die einzelnen Arten des Traceability gegeben wurde, ist es nun an der Zeit, sich etwas näher mit dem Meta-Modell zu beschäftigen. Der Bedarf eines besseren Verständnisses des Traceability ist weitgehend bekannt. Hierzu wurde von einigen Institutionen aufgezeigt, welche Punkte bei der Entwicklung eines solchen Meta-Modells zu beachten sind.

4.1 Eigenschaften des Meta-Modells

Laut der Veröffentlichung des Papers „Issues in the Development of a Model of Traceability" von den Autoren Ramesh und Edwards sind folgende Punkte zu berücksichtigen [9]:

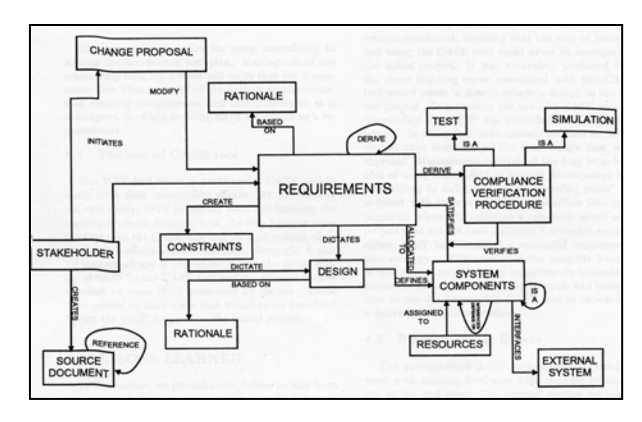


Abb. 6: Requirements Traceability Meta-Model [9]

4.1.1 Bidirektionales Traceability

Das bidirektionale Tracing schließt beide Richtungen ein: jenes nach vor und auch das zurück. Das Vorwärts-Tracing wird zur Verfügung gestellt, wenn jede Anforderung einen eindeutigen Designbestandteil referenziert. In anderen Worten gesagt, lässt Vorwärts-Traceability einen wirklich sehen, wo Anforderungen im Endsystem verwirklicht werden.

Das Rückwärts-Tracing wird hingegen dann zur Verfügung gestellt, wenn jede Anforderung von einem eindeutigen Designbestandteil referenziert wird.

4.1.2 Kritik von Anforderungen

Ein sinnvoller Weg, kritische Anforderungen zu identifizieren ist es, wenn jene mit den zentralen Aufträgen des Systems in Beziehung gestellt werden. Geschäftsprozesse oder Aufträge, die Anforderungen erzeugen, könnten gekennzeichnet und Anforderungen im Bezug auf solche Prozesse ausgewertet werden, um eine Klassifikation zu ermöglichen. Zum Beispiel sollte Traceability Punkte aufzeichnen, wie Anforderungen eingelangt sind.

4.1.3 Grundprinzip des Designs

Eine grundlegende Komponente des Traceability ist ihr Design. Traceability Verbindungen, um jene Grundprinzipien zu repräsentieren, würden das „Warum" bzw. den Grund von Designentscheidungen erläutern. Das Nachverfolgen entlang eines Designobjektes und das Verstehen, wie sich eine Änderung auf welches Objekt auswirkt, ist lebensnotwenig bei der Wartung des Systems.

4.1.4 Projektverfolgung "vs." Projektmanagement

Requirements Traceability kann in der Projektverfolgung und im Projektmanagement sehr sinnvoll und produktiv genutzt werden. Während der Systemdefinition und einzelnen Folgephasen, gewährleistet Traceability, dass keine Systemanforderungen verloren gehen bzw. alle eingehalten werden. Die Projektmanager könnten „Verbindungen" wie Status, Vollständigkeit der Daten und Autorisierung zwischen verschiedenen Systembestandteilen für die Terminplanung, Fortbestand und Sicherheit verwenden.

4.1.5 Verantwortlichkeit

Der Großteil, der Traceability verwendet, tut dies um den Verantwortlichkeitsbereich zu verbessern. Die Verwendung von Traceability legitimisiert beispielsweise die Kommunikation zwischen dem Designer einer Systemkomponente oder hilft beim Verständnis der Leistungsfähigkeit. Der Gebrauch von verantwortlichkeitsrelevanten Informationen als Mittel für Leistungsbewertung ist hier nicht angebracht.

4.1.6 Humanware

Humanware ist bei Projekten immer ein sehr kritischer Bestandteil. Es ist wichtiger denn je, wie Humanware und die Anforderungen an ein System in Verbindung stehen, da es die Verantwortlichkeit für eine Anforderung zu einer Person aufspüren und miteinbeziehen kann. Ein Beispiel für so eine Verbindung wäre die Systemfunktionalität, die ja von einer Person ausgeführt wird bzw. wurde.

4.1.7 Dokumente und Handbücher

Dokumenten-Traceability beendet die Beziehung zwischen 2 Dokumentsegmenten: sprich ein bestimmtes Dokument ist in Übereinstimmung mit einem vorhergehenden Dokument, mit dem es eine Art Verhältnis hat. Dokumenten-Traceability garantiert weiters, dass alle Komponenten eines Dokumentes eine „verwandte" Komponente in einem anderen Dokument besitzen. Konsistenz- und Vollständigkeitsbedingungen werden innerhalb von und über alle Dokumente angewandt.

4.1.8 Horizontales und vertikales Traceability

Vertikales Traceability referenziert die „Assoziation zwischen verschiedenen Software Life-Cycle Objekten (SLCO)". Ein Beispiel dafür wäre die Beziehung zwischen einem Anforderungsstatement und einem Designstatement. Horizontales Traceability referenziert die „Assoziation zwischen gleichen SLOC. Ein Beispiel für diese Traceabilityart wäre eine Eltern/Kind Beziehung zwischen aufgespalteten Datenflussdiagrammen einerseits und andererseits abgeleitete Beziehungen zwischen Anforderungsstatements.

4.1.9 Automatische Unterstützung für Traceability

Automatischer Support für Traceability kann sehr wertvoll für große bzw. komplexe Systeme sein. Wenn die Unterstützung manuell durchgeführt wird, sind die Aufgaben zeitraubend und fehleranfällig. Außerdem werden Fähigkeiten der Benutzer, Traceability-Daten, zu analysieren durch den schieren Datenbestand begrenzt.

Als Schlussfolgerung ergibt sich daraus, dass umfassende „Teil-Modelle" – darunter werden die 9 Eigenschaften des Meta-Modells verstanden - enthalten sein sollten, da es die verschiedensten Stakeholder in der Entwicklung eines Systems unterstützt. Solche Systeme, die essentielle Fähigkeiten eines Systementwicklungsprozesses erfassen können, sind notwendige Komponenten für einen sinnvollen Einsatz von Traceability-Schemata.

4.2 Wer zieht Nutzen aus dem Traceability Modell?

Nachdem nun verdeutlicht wurde, welche Eigenschaften solche Modelle besitzen sollten, um den SEP

zu erleichtern, möchte ich nun kurz auf die Personen eingehen, die am SEP arbeiten und die Nutzen aus dem Modell ziehen können. Eine Studie von WST[1], fokussiert am Gebrauch von Requirements Traceability, kam zum Schluss, dass folgende Parteien im Softwareentwicklungsprozess Nutzen aus einem Traceability-Modell ziehen können [14]:

4.2.1 Top-Management

Das Top-Management von WST sieht den Gebrauch von Requirements Traceability als ein Muss für das Überleben. Übereinstimmend kam das Management zur Aussage, dass Anforderungen keine Wahl sind, sie sind notwendig und dass es wichtig sei, den Kunden glücklich zu machen. Traceability stellen Kundenzufriedenheit sicher, indem wir ihm dokumentierte Mittel zur Verfügung stellen, durch die der Kunde überprüfen kann, ob alle angegebenen Anforderungen entsprechen und er somit sieht, dass der Job komplett ist.

4.2.2 Projekt-Management

Das Projektmanagement glaubt, dass der korrekte Gebrauch des Traceability Mittel zum Zweck ist, um die volle Steuerung zu haben und zu zeigen, dass alles läuft. Während der Designphase verwendet der Projektmanager das Traceability um dem Top-Management die Möglichkeit der Nachvollziehbarkeit zu gewährleisten. Es hilft ihm auch, Projektverzögerungen frühzeitig zu erkennen und somit gegenzusteuern.

4.2.3 Systemdesigner und Systemtechniker

Der bedeutendste Gebrauch von Traceability während eines Projektes wurde durch das "Notizbuch des Systemdesigners" beobachtet. Die gesammelten Daten, die in freier Textform notiert wurden, erfordern vom Systemdesigner ein großes Maß an Disziplin, um rational zu erklären, warum das System designt wurde wie es ist. Diese Informationen könnten Unschätzbares während der Life-Cycle-Wartung und bei der Entwicklung ähnlicher Systeme prüfen. Obgleich das Projekt nicht in dem Stadium der Wartung ist, sieht der Systemplaner weitgehend die Veränderungen an den Codemodulen und -unterlagen voraus.

[1] WST – Weapon System Technology Support Branche, eine amerikanische DoD Organisation, an der McClellan Air Force Base

4.2.4 Tester

Die Systemtester planen die Verwendung von Traceability, wenn sie den Plan für den Akzeptanztest schreiben. Unter der Verwendung des Traceability-Tools, überprüfen die Tester, ob der Akzeptanztest alle an das System gestellte Anforderungen erfüllt, um somit jene Vollständigkeit und Funktionalität sicherzustellen, die vom Kunde gefordert wurde.

Wie zu entnehmen ist, verwenden bei WST alle Requirements Traceability als ein wichtiges Qualitätsmanagementwerkzeug. Vom Top-Management bis hinunter zum Systemwartungspersonal: alle glauben daran, dass RT für einen erfolgreichen Abschluss unentbehrlich sei.

5. Qualität und Requirements Traceability

Die Nachfrage von Qualitätskontrolle, Nachvollziehbarkeit und Dokumentation steigt ständig an. Wie sich aus der Arbeit schon herauskristallisierte, ist es schwierig, Benutzeranforderungen mit notorischer Genauigkeit zu erreichen, da sie häufig instabil sind und über die Zeit immer wieder neu definiert werden. Außerdem neigt die Benutzerzufriedenheit eine kollektiv- und subjektive Angelegenheit zu sein. Qualitätsanforderungen werden im Allgemeinen aus externen Standards oder Politikdokumenten importiert.

Qualität hängt mit der Übereinstimmung der Systemspezifikation, mit den Systemeigenschaften und mit der Kundenzufriedenheit zusammen. RT hat Einfluss auf jeden dieser Faktoren. Ursprünglich wurde RT nur zur Überprüfung der geforderten Systemfähigkeiten und -eigenschaften verwendet. IT-Organisationen, die Anforderungen nicht nachvollzogen, dachten häufig, dass es eine bloße Übung in Gründlichkeit und Vollständigkeit sei und die Qualität in gelieferten Eigenschaften und Funktionalitäten ausgedrückt ist. Jedoch durch das Aufkommen der Anforderungs-Tools, reifte RT, um so Projekt-, Auswirkungs-, Analyse- und Änderungs- sowie Defektmanagement zu stützen und weiters eine Verbesserung der Fertigungsprozesse und Teamkommunikationen zu erzielen. Heute bietet das Anwenden von Traceability ein hohes Niveau der Projektsteuerung und der sicheren Qualität an, die durch alle möglichen anderen Mittel schwierig zu erzielen ist [15].

5.1 Kosten und Nutzen von RT

Abhängig davon, wie Unternehmen die Traceability-Informationen einsetzen, kommt es zu einem dem-

entsprechenden Nutzen. An einem Minimum können Projekt-, Entwicklungs- und Prüfungsgruppenleiter prüfen, ob alle Kundenanforderungen eingeführt und geprüft worden sind und ob Qualität, zuverlässige Systemfähigkeiten und -eigenschaften sichergestellt sind. Das Anwenden von RT gibt Organisationen sogar erheblichere Vorteile: erhöhte Qualität, verringerte Kosten und Verbesserung des Gefahren- und Projektmanagements für Projekte - gegenwärtige und zukünftige[15].

Traceability lässt uns alle Produktanforderungen sehr früh im Life-Cycle der Entwicklung zuteilen und somit erheblich Geld sparen lässt. Die Kosten die sonst beim „Warten" entstehen, bis die Integration und System-Test Phase abgeschlossen ist, um Defekte zu beheben, ist ein fixer Bestandteil und ist 30 mal stärker als bei der vorgesehenen Ausgangsphase [2].

6. Conclusio

Beginnen möchte ich das Fazit mit 2 Zitaten, die die Relevanz und Notwendigkeit von Requirements Traceability veranschaulichen: "You can't manage what you can't trace" und "Requirements Traceability can help in managing change and cost for all types of projects [1]"

Wie dieser Arbeit zu entnehmen ist, wird der Requirements Traceability „Mythos" als täglicher, lebenswichtiger Betrieb angesehen, da der Mechanismus des Tracings die Aufgaben der Life-Cycle Wartung enorm erleichtert und hilft, Prozesswissen in SW-Entwicklungsorganisationen zu erzeugen, speichern, konsistent zu halten, sie aufzubereiten und immer verfügbar zu halten. Auf Grund der Arten des RT wurde veranschaulicht wo die Probleme liegen und wie diese verbessert werden können.

Die Einführung eines erfolgreichen Requirements Traceability in einer Organisation ist eine große Herausforderung. Die Gründe dafür sind zahlreich: verschiedene Sichtweisen, Modelle, Zwecke, Qualitätsattribute und ihre Abhängigkeiten der Anforderungen untereinander. Wie schon erwähnt wurde, ist RT eine leistungsfähige Methode, die eine Datensammlung unterstützt, die genau auf die Ziele einer Organisation ausgerichtet ist. Diese Arbeit soll aufzeigen, dass der Einsatz von RT in einer Organisation sinnvoll ist.

Referenzen

[1] O.C.Z. Gotel and A.C.W. Finkelstein, "An Analysis of the Requirements Traceability Problem", Proceedings of the First International Conference on Requirements Engineering, IEEE, Colorado Springs, April 1994, pp. 91-101.

[2] R. Watkins and M. Neal, "Why and How of Requirements Tracing", IEEE Software, Juli 1994, pp. 104-106.

[3] M. Jake, "Requirements Tracing", Communications of the ACM, December 1998/Vol. 41, No. 12, pp. 32-36.

[4] K. Pohl, "Enabling Requirements Pre-Traceability", Proceedings of the IEEE Intl. Conference on Requirements Engineering (ICRE'96), Colorado, 1996.

[5] K. Pohl, „Process Centered Requirements Engineering", RSP marketed by J. Wiley & Sons Ltd., UK, 1996.

[6] O. Gotel, and A. Finkelstein. An analysis of the Requirements Traceability Problem, Imperial College Department of Computing Technical Report TR-93-41, 1993.

[7] O. Gotel and A. Finkelstein, "Contribution Structures", Proceedings of the Second IEEE International Symposium on Requirements Engineering (RE'95), York, IEEE Computer Society Press, 100-107, 1995.

[8] K. Pohl, R. Dömges, P. Haumer, R. Klamma, K. Weidenhaupt, „PRO-ART: Erfassung und Verwaltung von Anforderungshistorien", RWRH Aachen, Informatik 5, Germany

[9] B. Rahmesh, M. Edward Ramesh, "Issues in the Development of a Requirements Traceability Model", Proceedings of the IEEE International Symposium on Requirements Engineering, San Diego, California, Jan. 4-6, pp. 256-259

[10] C. Kenny "Requirements Traceability", Cmpt 856 Project, Jan. 1996
Quelle:
http://citeseer.ist.psu.edu/cache/papers/cs/21218/http:zSzzSzwww.cs.usask.cazSzhomepageszSzgradszSzcab130zSz856zSztrace.pdf/kenny96requirements.pdf

[11] A. Finkelstein, "Tracing Back from Requirements", in IEE. (1991). Tools and Techniques for Maintaining Traceability During Design, IEE Colloquium, Computing and Control Division, Professional Group C1, Digest No.: 1991/180, pp. 7/1-7/2.

[12] K. Pohl, "PRO-ART: Enabling Requirements Pre-Traceability", RWTH Aachen, Informatik V, 1995

[13] A. Dahlsted, "Requirements Engineering", Chaper 11, Department of Computer Science, University of Skövde
Quelle:
http://www.ida.his.se/ida/kurser/informationsystems_engineering/kursmaterial/forelasningar/Chapter11_2003.pdf

[14] B. Ramesh, T. Powers, C. Stubbs, M. Edward, "Implementing Requirements Traceability: A Case Study", In Proc. of the IEEE Int. Symposium On Requirements Engineering (RE'95), York UK, March 1995, pp. 89-95.

[15] Quelle: http://www.computerware.com/dl/reqtrace.pdf

Spannungen zwischen Benutzergewohnheiten und neuer Bedienbarkeit

Werner Sühs
9855851
wsuehs@edu.uni-klu.ac.at

Abstract

Die Gewohnheiten eines Benutzers werden über Jahre der Arbeit mit Computern und Programmen hinweg geprägt. Es entsteht eine Fixierung auf die Art und Weise, wie ein gewohntes Programm seine Arbeit erledigt und zu bedienen ist. Wird nun ein Wechsel des Programms vorgenommen, sei es aus technischen Gründen, weil das bisherige Programm neuen Anforderungen nicht gerecht wird, oder sei es aus finanziellen oder anderen Gründen, stehen die Benutzer meist vor dem Problem, mit einer veränderten Schnittstelle klar kommen zu müssen. Daraus können Spannungen entstehen, wenn eine Gewöhnung an das neue Programm nicht eintritt oder sich nur sehr langsam entwickelt. In dieser Arbeit wird auf die Grundlagen solcher Probleme eingegangen, um dann die auftretenden Spannungen zu betrachten und Lösungsmöglichkeiten zu suchen und aufzuzeigen. Ich werde in dieser Arbeit auch versuchen, verständliche Beispiele zu bringen, um die Problematik und ihre Konsequenzen zu verdeutlichen.

1. Einleitung und Motivation

1.1. Motivation

Wer von uns hat sich noch nie mit Software konfrontiert gesehen, die ihm oder ihr einfach unverständlich zu sein schien. Nicht dass diese Software explizit falsch oder fehlerhaft gewesen wäre, sie widersetzte sich einfach nur dem Verständnis und der Assoziation mit bisherigen Erfahrungen. Wir sind nicht gewohnt, derart häufig eine Veränderung der Benutzerschnittstelle zu erleben, in anderen Bereichen des Lebens, selbst in anderen Gebieten der Technik bleibt die Art und Weise, wie sich Dinge benützen lassen, über einen lagen Zeitraum unverändert. Kaum jemand würde beispielsweise eine Veränderung in der Bedienbarkeit eines Autos so einfach hinnehmen wie eine Veränderung einer Benutzerschnittstelle in der Computerwelt. Eine derartige Belastung im Lernaufwand kann schnell zu Problemen und Ablehnung führen. Ein Übermaß an Details wirkt verwirrend und verhindert, dass ein konzeptuelles Modell des Systems entwickelt wird. Ist der Benutzer über einen längeren Zeitraum nicht in der Lage, die ihm gestellten Aufgaben zu erledigen, kommt es zu Frustration. Diese Probleme können die Arbeitsleistung reduzieren oder sogar zu einer Nutzungsverweigerung führen. [1]

1.2. Ein einleitendes Beispiel

Vor etwa 20 Jahren filmte ein Forscher mit versteckter Kamera führende Xerox- PARC Computerwissenschaftler beim Versuch, das neue Kopiergerät zu benutzen. Das Ergebnis: Die Benutzer wurden zunehmend frustrierter und wütender, weil der Kopierer nicht das produzierte, was sie wollten, und sie sich selber nicht zu helfen wussten. Dieses Video zeigte den Entwicklern zum ersten Mal auf, dass die von ihnen nach allen Regeln der Kunst entwickelten Kopiergeräte aufgrund der zu komplizierten Benutzeroberfläche nicht bedienerfreundlich waren. Unter Berücksichtigung dieser Beobachtungen wurde eine neue Oberfläche entworfen. Das Resultat: Die durchschnittliche Zeit zur Behebung eines Papierstaus wurde von 28 Minuten auf 20 Sekunden reduziert. [2]

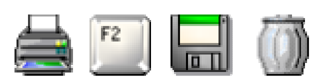
2. Grundlagen:

2.1. Werkzeuge der Benutzerschnittstelle:

Eine **Analogie** (griechisch analogos: proportional, entsprechend, verhältnismäßig) ist die Ähnlichkeit zwischen zwei Strukturen, die aber nicht die gleiche Entstehungsgeschichte haben. Wenn sich etwas analog zu etwas anderem verhält, kann man vom Verhalten des einen auf das Verhalten des anderen schließen. Analogien werden eingesetzt, um Vertrautheit mit einem Konzept auf ein anderes zu übertragen. Ein Beispiel wäre das heute weit verbreitete Konzept von Drag and Drop, jedem Benutzer ist intuitiv

verständlich, wie man einen Gegenstand nimmt und bewegt.

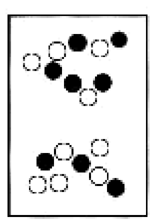
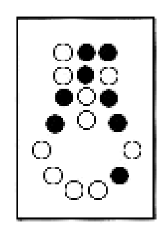
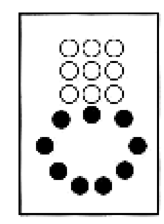
Ein **Piktogramm** (lateinisch pictum - Bild, griechisch gráphein - schreiben) ist ein einzelnes Bildsymbol, das eine Information durch vereinfachte grafische Darstellung vermittelt. Piktogramme waren die Vorläufer vieler früher Schriftzeichen und wurden später häufig zu Logogrammen weiterentwickelt. Einige Piktogramme werden heute noch im Japanischen bzw. Chinesischen verwendet. Ein Piktogramm sollte kultur- und bildungsneutral sein, also für Menschen der ganzen Welt und unterschiedlicher Bildung verständlich sein. Es darf keine Tabus verletzen, das heißt keine religiöse, sittliche oder rassistische Diskriminierung darstellen. Ein Piktogramm muss lesbar sein und die Information leicht zugänglich machen.



Der Ausdruck **Icon** (deutsch: Ikone griechisch: eikon Bild) bezeichnet im Computerbereich ein Piktogramm oder eine kleine Grafik. Icons sind Versinnbildlichungen von Befehlsfolgen und gestatten dem Anwender, Programme mit geringem Vorwissen einzusetzen. Hinter dem Icon verbirgt sich ein Link, der ein Programm ausführt oder es ermöglicht, zwischen Webseiten zu navigieren. Im Gegensatz zu Blickfängern, die Aufmerksamkeit binden sollen, sind Icons für den Benutzer nützlich und inhaltlich sinnvoll. Das in einem Icon verwendete Bild muss, wie es für ein Piktogramm üblich ist, für alle Benutzer verständlich sein. Unterschieden werden Icons in Ähnlichkeitsicons (Bild analog zu Bedeutung), Exemplarische Icons (typisches Beispiel), Symbolische Icons (Abstraktion auf höherer Ebene) und beliebige Icon (ohne Assoziation).

Die Software **Ergonomie** beschäftigt sich mit dem menschen- bzw. aufgabengerechten Design des Mensch-Computer Dialogs. „Software Ergonomie" befasst sich mit der Analyse, Gestaltung und Bewertung der Arbeit des Menschen an oder mit rechnergestützten 'dialogfähigen' Systemen, soweit diese durch die Software bestimmt sind, mit dem Ziel einer „menschengerechten Gestaltung des Arbeitsmittels". Diese erst seit ca. 20 Jahren existierende interdisziplinäre Fachdisziplin gründet sich zum großen Teil auf Erkenntnisse der Psychologie, Physiologie, Arbeitswissenschaften, Informatik bzw. Grafikdesign. [3]

Für die benutzergerechte **Anordnung** von Bildelementen ist die Gestaltpsychologie bedeutsam, die bildliche Szenen möglichst einfach interpretierbar darstellen will. Nach dem Prägnanzprinzip setzt sich immer eine Gliederung von Elementen durch, die der menschlichen Neigung zu Einfachheit, Symmetrie, Regelhaftigkeit und Geschlossenheit entgegenkommt. Kleine Figuren sind vor einem großen Hintergrund schneller identifizierbar als umgekehrt, dunkle Figuren werden vor einem hellen Hintergrund schneller erkannt, räumlich zentrale Elemente fallen früher auf als periphere und symmetrische und horizontal/vertikal ausgerichtete Figuren erscheinen eindeutiger. Räumlich oder zeitlich eng zusammen liegende Elemente werden leichter als Einheit betrachtet, genau wie einander ähnliche Elemente (Farbe und Größe sind hier bedeutsamer als Form). Bildet die Anordnung ein kontinuierliches Muster, werden die Elemente ebenfalls als zusammengehörig erkannt. Werden die genannten Anordnungsstile vermischt, können sie sich unter Umständen gegenseitig schwächen. [3]

Die **Anordnung** der Elemente sollte auch dem logischen Arbeitsablauf Rechnung tragen, und zwar noch bevor sie nach Wichtigkeit und Häufigkeit gruppiert werden, weiters sollte die Erwartungstreue eingehalten werden, gemäß dem Prinzip „Kennt man einen, kennt man alle". [1] Nicht auszudenken wäre die Verwirrung, würde für die drei Buttons in Windows Fenstern (Minimieren, Maximieren, Schließen) die Reihenfolge verändert.

Bei der Verwendung von **Farben** sind obige Gruppierungsregeln von hoher Bedeutung, meist werden Farben eben zum Zweck der Gruppierung eingesetzt. Weiters spiegeln Farben die Dringlichkeit von Elementen wieder, auffällige Farben werden eingesetzt, um Aufmerksamkeit zu erregen. Das Erlernen einer Benutzeroberfläche wird durch Farbassoziationen ebenfalls erleichtert.

Um das **Verhalten** eines Systems zu charakterisieren, kann man den Zeitpunkt, in dem der Benutzer die Kontrolle hat, als Zustand beschreiben. Diese Zustände lassen sich nun gemeinsam mit den Aktionen des Computers und den Übergängen zwischen Zuständen

und Aktionen zu einem Diagramm zusammenfassen. Denert [4] spricht hier von „Schematischen Interaktionsdiagrammen“. Ausgehend von Zuständen, in denen der Computer wartet, wird durch betätigen einer virtuellen Taste ein Zustandsübergang zu einer Aktion ausgelöst, wenn diese erledigt ist, gelangt das System wieder in einen Zustand. Als virtuelle Taste wird jede Aktion des Benutzers gesehen, sei es nun eine Taste des Keyboards, ein Button, eine Menüauswahl oder ein Kommando.

Grundsätzlich sollte das **Verhalten** eines Programms dem Ablauf entsprechen, in dem Benutzer ihre Arbeit erledigen, durch den Computer hinzukommende Aspekte sollten den Benutzer nicht behindern. Jeder Benutzer hat bestimmte Erfahrungen, wie bestimmte immer wieder auftauchende Aktionen ablaufen, beispielhaft hierfür wäre nicht nur, wo man ein Dokument speichern kann, sondern auch wie viele Klicks dazu nötig sind. Ein Programm sollte darüber hinaus für den Benutzer steuerbar sein, er muss die Geschwindigkeit des Arbeitsablaufs und die Reihenfolge der Aktivitäten beeinflussen können. [1]

Im Rahmen des Verhaltens eines Systems von Bedeutung ist die **Navigation**. Die Navigation beginnt bei einem System bei einem bestimmten Einstiegspunkt. Im westlichen Kulturkreis wird anfängliche Information zuerst links oben gesucht, im Arabischen und Asiatischen rechts oben. Die Navigation sollte durch Gruppierung der Elemente und klare Grenzen vereinfacht werden. Des Weiteren sollte auch die Navigation per Tastatur (Tabulator Taste) unterstützt werden. [1]

Um die Benutzer mit Informationen über die Aktivitäten des interaktiven Systems zu versorgen, werden sie konstant mit **Feedback** versorgt. Dem Benutzer sollten ständig seine bisher getroffenen Entscheidungen sowie deren Auswirkungen ersichtlich sein. Feedback zur Verfügbarkeit kann durch Markierung oder Ausblendung von Elementen geliefert werden, die Verfügbarkeit des ganzen Systems kann mit einer Sanduhr oder einem Fortschrittsbalken verbildlicht werden. Textuelles Feedback sollte klar verständlich und höflich sein, des weiteren ist es für den Benutzer nützlicher, bei einer Fehleingabe zu erfahren, was er besser machen kann, anstatt zu erfahren, was er falsch gemacht hat.

2.2. Psychologische Faktoren:

Für die Entwicklung eines Programms ist es lebenswichtig, die **Erwartungen** der Benutzer richtig einzuschätzen, selbst für den Fall, dass das Programm auf anonyme Benutzer abzielt. Diese Erwartungen können Bearbeitungsreihenfolge von Prozessen, Toleranz bei Wartezeiten, strukturierten Entwurf, Verständlichkeit, Konformität mit ähnlichen Programmen und vieles mehr umfassen. Auch negative Erwartungen sind zu bedenken, also Ereignisse, die auf keinen Fall eintreten dürfen. Generell sollen Erwartungen klar ausgedrückt werden, da Fehler bei ihrer Erfassung zu Ungunsten der Benutzer und damit zu Ungunsten des Projekts gehen. Darüber hinaus ist es sinnvoll, bewusst Erwartungen, die nicht erfüllt werden können, auszuschließen. Einerseits um die Grenzen des Produkts zu klären und ihm damit eine klarere Identität zu geben, andererseits, um Enttäuschungen, die auf unausgesprochenen Erwartungen beruhen, vorzubeugen. Weinberg [5] spricht von einem Prozess bestehend aus vier Schritten. Zuerst werden möglichst repräsentative Benutzer nach ihren Erwartungen gefragt, daraus wird dann eine Liste erstellt. Diese Liste wird dann generalisiert und zuletzt werden die generalisierten Listeneinträge drei Gruppen zugeordnet: möglich, verschieben, unmöglich. Die Elemente der ersten Gruppe werden umgesetzt, die der zweiten kommen auf eine Warteliste und die der dritten werden verworfen.

Um die Erwartungen der Benutzer richtig zu erfassen, ist die Kultur der **Kommunikation** von großer Bedeutung. Hierzu ist es nicht nur wichtig, mitzuteilen, sondern auch sicherzugehen, dass die Nachricht empfangen und verstanden wurde. Feedback ist von großer Bedeutung und wer kein Feedback von den Benutzern einholt, kann niemals Gewissheit haben, wo er gerade steht. Unterscheiden kann man Kommunikation in schriftlich bzw. mündlich und formell bzw. informell, was davon zur Anwendung kommt, hängt von der Absicht des Mitteilenden und von seinem Publikum ab. Kommunikationsbarrieren können mangelnde Vorbereitung, Körpersprache, Akzent oder anderes sein und auf die potenzielle Fähigkeit des Gegenübers, dies auszugleichen, sollte man sich nicht verlassen. [6]

Wird eine **Veränderung** an einem System vorgenommen oder ein System gegen ein anderes ausgewechselt, ist sowohl mit positiven als auch negativen Reaktionen zu rechnen. Bei einfacher Betrachtungsweise gibt es Benutzer, die Verbesserungen begrüßen und solche, die lieber bei Bewährtem bleiben. Aber darüber hinaus gibt es bei jeder Veränderung immer Gewinner und Verlierer, selbst die revolutionärste Verbesserung wird auf irgendeinem Gebiet eine Verschlechterung bedeuten. [5] Daraus folgt, dass es sinnvoll ist, alle möglichen aktiven und passiven Benutzergruppen zu betrachten

und herauszufinden, was sie gewinnen und verlieren. Weiters mag es sinnvoll sein, bewusst festzulegen, welche Benutzergruppen Verlierer sein sollen, beispielsweise soll Kindern der Umgang mit Steckdosen oder Einbrechern der Zutritt zu einem Gebäude erschwert werden. Genauso, wie Benutzer Veränderungen anregen, können aber auch Veränderungen Benutzer hervorbringen, wenn beispielsweise ein Labor in einem Krankenhaus schnellere Analysegeräte bekommt, wird die Zahl der Anfragen ebenfalls steigen.

Betrachtet man die menschliche **Wahrnehmung**, so ist die Grundlage die Aufnahmefähigkeit der Sinne. Das Auge kann nur auf einem kleinen fokussierten Bereich klar sehen und in der Peripherie des Blickfelds sinkt die Schärfe und damit steigt die Empfänglichkeit für Bewegungen. Auf einer höheren Ebene der Wahrnehmung spielt Vertrautheit eine Rolle und wir verstehen leichter Dinge, die uns bereits bekannt sind. So tendieren auch viele Entwickler von Software dazu, aufgrund ihrer eigenen Vertrautheit mit der Benutzerschnittstelle, diese als intuitiv zu betrachten. Doch intuitiv sind Signale immer nur für bestimmte Benutzergruppen, beispielsweise ist der Entwurf von international verständlichen Verkehrszeichen sehr schwer. Ein Beispiel für unterschiedliche intuitive Wahrnehmung ist die Assoziation von Farben in Japan, hier gilt Weiß als die Farbe der Trauer und des Todes und in unterschiedliche Regionen Afrikas steht Rot für Leben und in anderen für Trauer. Weiters zu erwähnen ist der Unterschied zwischen Orient und Okzident in der Frage, ob Schriften und folglich auch Benutzerschnittstellen ihren Ursprungspunkt auf der rechten oder auf der linken Seite haben.

Wenn ein Benutzer mit einem System interagiert, so tut er dies immer aufgrund eines **Gedankenmodells**. Zusätzlich zur allgemein bekannten Rolle der unmittelbar sichtbaren Komponenten bildet er sich ein Modell des Verhaltens der für ihn unsichtbaren Komponenten. Ist dieses Modell nun fehlerhaft und stimmt nicht mit der Realität überein, so kann der Benutzer auch nur unangemessen auf Ereignisse reagieren und wird Fehler begehen. Ein normaler Autofahrer hat ein Gedankenmodell von seinem Auto, er versucht die Zusammenhänge zwischen Lenkrad, Motor, Getriebe, Batterie, Reifen und allen anderen Komponenten zu verstehen. Lässt sich der Wagen einmal nicht starten, würden die meisten Fahrer annehmen, dass die Batterie leer ist. Tritt nun ein komplizierter Fehler auf, muss der Wagen zu einem Mechaniker gebracht werden, der ein unfangreicheres und korrekteres Gedankenmodell des Autos hat und so in der Lage ist, es zu reparieren. Benutzer eines neuen unbekannten Systems haben ein nur unklares Bild über die Vorgänge und betrachten es anfangs als eine Art Black Box. In ihrem geistigen Gepäck tragen sie Erwartungen und ältere Gedankenmodelle mit sich, die zur Bildung eines fehlerhaften Modells führen können. Fehler die nun begangen werden sind generell systematisch, da ja nicht zufällig vorgegangen wird, sondern nach einem klaren System, das eben nur fehlerhaft ist. Eingesetzt werden Gedankenmodelle, weil sie uns erlauben, die Zukunft vorherzusehen - wer das Licht seines Autos brennen lässt, weiß, dass am nächsten Morgen die Batterie leer sein wird. Außerdem helfen sie uns, die Ursachen von beobachteten Ereignissen zu finden und sie sagen uns, was zu tun ist, um ein bestimmtes Ergebnis zu erzielen. [7]

Meistens, wenn ein Benutzer einem neuen Programm gegenübersteht, hat er bereits Erfahrung in der Benutzung von Programmen dieser Art. Ist das Programm ein Folgeprogramm eines ihm bekannten, ist von Anfang an Vertrautheit gegeben. Handelt es sich hingegen um eine völlig neue Benutzeroberfläche, ist das **Erlernen** dieses Programms Voraussetzung für seine Benutzung. Auf dem Weg zum Lernen passiert eine Information zuerst das Kurzzeitgedächtnis, um dann in das Langzeitgedächtnis zu gelangen. Das Kurzzeitgedächtnis ist in der Lage, sich über einen Zeitraum von bis zu dreißig Sekunden 7 +/- 2 Dinge zu merken. Werden mehr Elemente aufgenommen, gehen die ältesten Elemente zugunsten der Neuen verloren. Das Langzeitgedächtnis im Gegensatz dazu bietet eine unbegrenzte Kapazität, dafür ist der Zugang dazu weniger verlässlich, die Speicherzeit variiert von acht Sekunden bis zu mehreren Jahren. Das Abrufen von Informationen aus dem Langzeitgedächtnis ist abhängig von Assoziationen, bei Erleben von mit einer Erinnerung in Verbindung stehenden Reizen fällt der Zugriff auf das Langzeitgedächtnis leichter. Dies lässt sich durch den Einsatz von vertrauten Bildern und Elementen nützen und in dem Entwurf von Benutzerschnittstellen werden vertraute Bilder wie Abbildungen von Disketten, Druckern oder Papierkörben eingesetzt.

Die **Zufriedenheit** der Kunden sollte aus Gründen des eigenen Überlebens jedem Entwickler am Herzen liegen. Sie unterliegt einem ständigen Wandel und muss gemessen werden, um die eigene Position zu bestimmen. Wichtig sind hier die Wiederholbarkeit von Tests und die Möglichkeit, Unzufriedenheit speziellen Bereichen und Benutzergruppen zuzuordnen. Gause und Weinberg [5] empfehlen hier, die Benutzer selbst die Testkriterien bestimmen zu lassen und Kommentare bei der Auswertung der Testbögen ernst zu nehmen. Vor allem Prototypen

eignen sich zum Testen von Zufriedenheit hervorragend, sie lassen Schlüsse auf das zukünftige Verhalten der Benutzer am fertigen System zu. Eine weitere Methode, um Zufriedenheit verlässlich zu bestimmen oder gar zu lenken ist die Einbindung der Benutzer nicht nur in die Spezifikation und den Test sondern auch in den Entwurf, siehe die ETHICS Methode, die in einem späteren Kapitel noch genauer beschrieben wird.

2.3. Beteiligte:

Die erste Frage, die zum Thema **Benutzer** zu klären ist, ist, wer diese Benutzer eigentlich sind. Zuerst muss zwischen Kunde und Benutzer unterschieden werden, denn der Kunde ist nur die Person oder Organisation, die zahlt, der Benutzer hingegen wird lange Zeit von dem Produkt betroffen sein. Sie unterscheiden sich in Erwartungen und Macht, denn obwohl der Kunde mehr Einfluss hat, da er ja zahlt, hat der Benutzer meist die höheren Erwartungen und wird sich im Gegensatz zum Kunden nicht immer mit der Erfüllung des Pflichtenhefts begnügen. Zusätzlich zu Kunden werden auch die Benutzer häufig in den Entwicklungsprozess einbezogen, anfangs bei der Anforderungsanalyse und am Ende bei Tests, darüber hinaus bei speziellen Methoden wie ETHICS sogar über die gesamte Dauer des Projekts hinweg. Dieses Einbeziehen ist ein zunehmender Trend und dies zu Recht, denn es ist viel leichter, die Wünsche der Benutzer zu erfüllen als ihre Bedürfnisse vorherzusagen, vorausgesetzt, die Benutzer sind auch willig sich zu beteiligen und erhalten von ihren Vorgesetzten die nötige Zeit dazu. Für das genaue Verständnis der Benutzer und ihrer Wünsche ist es nützlich, sie in unterschiedliche Gruppen einzuteilen. Möglichkeiten hier wären Studenten, Manager, Kinder, Behinderte, Frauen, Männer, Lehrer, Ärzte und viele mehr. Hat man alle Beteiligten erkannt, kann man festlegen, wen man wie behandeln will, sei es ihre Wünsche zu erfüllen, unwichtige Benutzer zu ignorieren und feindliche Benutzer unfreundlich zu behandeln. [5] Diese Einteilung ist zwar meist mangelhaft, da Benutzer sich selbst innerhalb dieser Gruppen stark individuell verhalten, aber eine unzureichende Einteilung ist immer noch besser als gar keine. Unterschiede finden sich auch in der Erfahrung der Benutzer, die sich nur schwer auf einer linearen Skala messen lässt. Einerseits ist hier zwischen allgemeiner Erfahrung mit Programmen und ihren Aufgaben und andererseits zwischen spezieller Erfahrung mit einem bestimmten Programm und ähnlichen Produkten zu unterscheiden. Diese Eigenschaften sind bei der Entwicklung von Programmen zu bedenken, zielt es auf Benutzer mit unterschiedlichen Fähigkeiten ab, sollte das Programm in der Lage sein, sich diesen anzupassen.

Entwickler neigen dazu, aufgrund ihres technischen und formalen Kontexts Anforderungen wörtlich zu nehmen. Es ist nicht immer direkter Kontakt zwischen Entwickler und Benutzer gegeben und so mag die Spezifikation die einzige Basis sein, auf der der Entwickler aufbauen kann. Werden spezielle Anforderungen gestellt, besonders im Bereich der Effizienz, neigen viele Entwickler dazu, diese mit eisernem Gehorsam auszuführen und zwar meist zu Lasten anderer Aspekte. Sagt man einem Entwickler beispielsweise, er solle Speicherplatz minimieren, so wird das Programm voraussichtlich auf dem Gebiet der Geschwindigkeit weniger leisten, als es könnte. Aufgrund ihres hohen Grades an Wissen über Programmentwicklung im Allgemeinen und spezielle Produkte sind Entwickler voreingenommen und neigen dazu, die Überzeugung in sich zu tragen, den Weg, den eine Entwicklung nehmen soll, zu kennen. Ungenauigkeiten bei Fragestellung oder Antworten in der Anforderungsanalyse geben Entwicklern Freiheiten, eigene Präferenzen einfließen zu lassen. Doch Entwickler sollten niemals versuchen, dem Benutzer das zu geben, was er braucht, sondern das, was er haben will, wenn ein Kunde nicht willig ist, sich beraten zu lassen, dann haben seine Wünsche Vorrang vor der Erfahrung des Entwicklers. Gibt es hingegen im Stadium der Anforderungsanalyse weder Kunden noch Benutzer, so bleibt dem Entwickler die Freiheit seine Vorstellungen umzusetzen und später die Notwendigkeit, eventuelle Käufer dafür zu begeistern. Voreingenommen ist ein Entwickler aber nicht nur in der Anforderungsanalyse, sondern auch in der Testphase, denn durch sein umfassendes Wissen über das Produkt ist er nicht mehr fähig, dessen intuitive Verständlichkeit zu beurteilen. Folglich ist es notwendig, unbefangene Tester mit der Aufgabe zu betrauen, aber für die Entwickler bedeutet dies, dass es für sie nützlich ist, die Benutzer, ihr Umfeld und grundlegende Elemente der kognitiven Psychologie zu kennen, um ihre Reaktion auf die Benutzerschnittstelle besser einschätzen zu können. Am sichersten für den Entwickler ist hier die Annahme, dass jeder einzelne Benutzer sich von allen anderen unterscheidet. [7]

3. Beispiele:

Ein Beispiel für ein Programm mit einer sehr komplizierten und schwer zu erlernenden Benutzeroberfläche ist **SAP**. Es handelt sich hierbei um ein sehr umfangreiches und mächtiges Verwaltungsprogramm, das über unzählige

Bildschirme und Masken verfügt. Konsistenz zwischen den einzelnen Masken ist gegeben, die Möglichkeiten, einzelne Masken an die Wünsche der Benutzer anzupassen besteht nicht. SAP ist ein sehr starres Programm und Fehleingaben lassen sich nicht mehr so leicht korrigieren. Um den Zugang zu den einzelnen Masken zu erleichtern, können diese mit vierstelligen Codes erreicht werden. All dies bürdet dem Benutzer einen immensen Lernaufwand auf, die Verwaltung all der Masken und Codes ist für Anfänger sehr verwirrend. Da ein intuitives Verständnis des Programms nicht gegeben ist, bedeutet dies, dass ein großer Lernaufwand bei der Übernahme des Systems zu bewältigen ist, Dokumentationen und Schulungen sind hier unverzichtbar. Die Firma SAP ist sich dessen bewusst und bietet umfangreiche Schulungen an, die Aufgrund der Mächtigkeit des Programms auch von vielen genutzt werden.

Im Vergleich zwischen **Linux** und **Windows** zeigt sich, welche Unterschiede Benutzerschnittstellen aufweisen können. Diese Unterschiede beruhen auf verschiedenen Weltbildern und wirken sich in allen Aspekten aus. Linux bietet die Freiheit, zu wählen, es handelt sich hier grundsätzlich um ein Kommandobasiertes Betriebssystem, das beliebig mit einer Graphischen Oberfläche erweitert werden kann. Windows bietet hingegen die Freiheit, nicht wählen zu müssen, und folglich ist hier alles standardisiert und leicht verständlich. Die Auswirkung auf die Interaktion beim Benutzen ist enorm und ein Wechsel zwischen diesen beiden Betriebssystemen stellt eine große Umstellung dar. [8]

WIMP Interfaces haben ihren Ursprung in den sechziger Jahren des letzten Jahrhunderts bei einem Mann namens Douglas Englebart und dem Human Augmentation Project bei SRI, die Abkürzung WIMP steht für Windows, Icons, Menus und Pointing device. Mit der Entwicklung der ersten Prototypen der heute allseits benutzten Maus entstand eine Oberfläche aus in der Größe veränderbaren Fenstern, Buttons, Popup-Menus, der Desktop-Metapher, einer objektorientierten Software Architektur und einer Entwicklungsbibliothek, die seit den achtziger Jahren die Computerwelt dominiert. Der Aufstieg von WIMP kam dann mit Steve Jobs und Bill Gates und der Macht der Firmen Apple Macintosh und Microsoft. Von da an war es in der Computerwelt üblich, die verschiedensten Objekte direkt auf dem Bildschirm mit der Maus zu manipulieren. Durch diesen bildlichen Einsatz von Metaphern ist das WIMP Interface intuitiv verständlich und heute jedem Benutzer vertraut. Standards, festgelegt von den Firmen Apple Macintosh und Microsoft, definieren die Form der WIMP Interfaces und garantieren, dass es für die Benutzer keine Überraschungen gibt. [9]

ETHICS steht für "effective technical and human implementation of computer systems" und wurde von Enid Mumford in Manchester entwickelt und dient der Beteiligung der Benutzer am Softwareentwurf und der Entwicklung von Software, die nicht nur effizient und effektiv ist, sondern auch benutzerfreundlich und humanistisch. Drei Grundziele werden durch Beteiligung der Benutzer am gesamten Entwicklungsprozess verfolgt: Erlernen des Systems, Akzeptanz der Software, und Kontrolle über die Veränderung in der Organisation, für die die Benutzer arbeiten. Hierfür werden folgende Schritte benötigt: Rechtfertigung der Veränderung, Abgrenzung der Systemgrenzen, Beschreibung des bisherigen Systems, Festlegen der Schlüsselziele und Aufgaben, Identifikation der Schwächen des bisherigen Systems, Analyse der Arbeitszufriedenheit, Analyse zukünftiger Entwicklungen, Spezifikation und Reihung von Effizienz- und Arbeitszufriedenheitsanforderungen, organisatorisches Design, Technische Aspekte, Vorbereitung eines detaillierten Arbeitsdesigns, Implementierung, Evaluierung. [10]

4. Lösungen:

Richtlinien und **Standards** haben sowohl für die Entwickler, als auch für die Benutzer ihren Nutzen. Sie stellen dem Entwickler bewährtes Wissen zur Verfügung und geben ihm somit Sicherheiten und ersparen ihm, unter Umständen, das Rad neu zu erfinden. Die kollektive Erfahrung aller Entwickler fließt in diese Richtlinien und Standards ein und so ist der einzelne Entwickler in der Lage, auf das Wissen aller zuzugreifen. Aber darüber hinaus haben diese Richtlinien und Standards noch eine weitere, auf den zweiten Blick sehr vorteilhafte Eigenschaft. Ihre eiserne Einhaltung engt den Entwickler ein und beraubt ihn vieler Möglichkeiten. Dies geht auf Kosten der Kreativität, und neue Ideen und Innovationen können schwerer umgesetzt werden. Aus der Sicht der meisten Entwickler eine unangenehme Sache, sie können sich nicht mehr völlig einbringen und fühlen sich unter Umständen eingeengt. Aber für die Benutzer hat diese Eigenschaft einen immensen Nutzen, durch die eingeschränkte Handlungsfreiheit der Entwickler sinkt die Zahl an Überraschungen, die ihnen bevorstehen könnten, und eine schnelle Vertrautheit mit einem neuen Produkt ist um vieles leichter zu erlangen. Für den Fall, dass die Anzahl der Benutzer eingeschätzt werden kann, und diese dem Entwickler in der Anforderungsanalyse zur Verfügung stehen, mag dies

von geringerer Bedeutung sein, da ein qualitativ hochwertiger Prozess der Anforderungsanalyse ebenfalls Überraschungen vermeiden kann, aber wenn es sich um ein Produkt für den freien Markt handelt, das nicht für einen bestimmten Benutzerkreis vorgesehen ist, steht in der Anforderungsanalyse niemand zur Verfügung, der Wünsche äußern kann, und hier ist der Einsatz von Richtlinien und Standards eine Absicherung, um einer späteren Akzeptanz näher zu kommen. Beispiele für Richtlinien sind Kompatibilität, sowohl zu Vorgängern als auch zu Konkurrenzprodukten, und Kompatibilität zur Art und Weise, wie ein Benutzer seine Aufgaben zu bewältigen versucht. Programme sollten konsistent sein, was bedeutet, dass Designs und Muster sich durch das ganze Programm ziehen und in jeder Maske wieder zu finden sind. Es sollten stets vertraute Muster Verwendung finden und eine Benutzerschnittstelle sollte unkompliziert sein, selbst wenn das Programm viele Funktionalitäten zur Verfügung stellt. Benutzer sollten in der Lage sein, so viel wie möglich direkt zu manipulieren und das System sollte ihnen das Gefühl geben, dass sie die Kontrolle über die Situation haben. Entsprechend dem Slogan „What you see is what you get“ sollte der Output von Interaktionen mit dem System immer im Zusammenhang mit dem aktuellen Inhalt des Bildschirms stehen. Ein Programm sollte sich flexibel auf die Wünsche des Benutzers einstellen und es sollte stets direkte Rückmeldungen nach einem Input geben, um dem Benutzer den Erfolg seiner Aktion anzuzeigen. Des Weiteren sollte ein Programm jegliche technische Details vor dem Benutzer verbergen und ihn möglichst vor unangenehmen Konsequenzen schützen. [7] Es ließen sich unzählige weitere Richtlinien und Standards finden, hier sei auf die internationalen Normen verwiesen.

Um auf die unterschiedlichen Erfahrungsstufen und persönlichen Präferenzen der einzelnen Benutzer eines Programms einzugehen, ist **Customizing** ein hilfreicher Weg. Wenn Benutzer in der Lage sind, das Programm nach ihren Wünschen zu gestalten, steigen sowohl Akzeptanz als auch Effizienz. Es kann Benutzern freigestellt werden, welche Art sie bevorzugen, das System mit Input zu versorgen, sei es per Maus, Keyboard, Touchpad oder auf anderen Wegen. Weiters können Benutzer in den heute gebräuchlichen Desktops Objekte nach ihren Bedürfnissen anordnen, Fensterpositionen und deren Größe ändern und Farben und Hintergrundbilder wählen. Es sollte möglich sein, zwischen verschiedenen Intensitäten der Interaktion zu wechseln, je nachdem wie vertraut ein Benutzer mit einem Programm ist, da Anfänger den Wunsch haben, alles leicht verständlich vorzufinden und Profis schnell Befehle eingeben wollen. Beispielhaft hierfür wären Einstellungen für Anfänger und Fortgeschrittene oder die Möglichkeit, zwischen graphischer Benutzeroberfläche und Kommandosprache zu wechseln. Aber Customizing muss sich nicht nur darauf beschränken, dem Benutzer Möglichkeiten zu geben, die Benutzerschnittstelle seinen Bedürfnissen anzupassen. Ein Programm kann auch das Verhalten eines Benutzers beobachten und aus seinen Aktionen auf sein Gedankenmodell und dessen Ausgereiftheit schließen. Meint das Programm, den Benutzer und sein Gedankenmodell richtig einschätzen zu können, kann es nun korrigierend reagieren und die Interaktionstiefe der Benutzerschnittstelle seinen Fähigkeiten anpassen. Ein derartiges Verhalten des Programms ist natürlich nur dann akzeptabel, wenn der Benutzer einverstanden ist und es muss jederzeit auch für unerfahrene Benutzer ein- und ausschaltbar sein. Wenn ein Benutzer nun an Erfahrung gewinnt, wird sich das Programm anpassen und ein schnelleres und effizienteres Bedienen ermöglichen. Voraussetzung dafür ist selbstverständlich, dass sich die Benutzer des Programms nicht ständig ändern. Customizing, ob nun aktiv oder passiv, ist für jedes Projekt, ob mit Benutzerbeteiligung oder ohne, nützlich, da einzelne Benutzer sehr individuelle Wünsche haben und so ein Maximum dieser erfüllt werden kann.

Evolutionäre Entwicklung versucht, durch ständiges Weiterentwickeln und darauf folgendem Erproben, ein Programm Schritt für Schritt aufzubauen. Jede Neuerung wird sogleich in der natürlichen Umgebung erprobt und sofort auf weitere Entwicklungsmöglichkeiten untersucht. Eine alte und intuitive Herangehensweise an evolutionäre Entwicklung ist „trial and error“, nach Fertigstellung eines Programms oder eines Teils davon wird dieses sofort gestestet und die Fehler behoben. Bei inkrementeller Entwicklung eines Programms spricht man bei den Zwischenstationen von Prototypen. Ein derartiges Vorgehen eignet sich besonders, wenn Änderungen der Anforderungen zu erwarten sind oder das endgültige Ergebnis am Anfang noch nicht klar ist und sich erst im Laufe des Entwicklungsprozesses ein Ziel herauskristallisiert. [11] Anwendung findet es überall, wo Benutzer und Kunden eng in den Entwicklungsprozess eingebunden sind, ein Beispiel hierfür wäre Extreme Programming, hier wird ein Wunsch des Kunden umgesetzt um ihm dann das Ergebnis zu präsentieren und dann weitere Wünsche in Angriff zu nehmen.

Reuse ermöglicht es Programmierern, bewährte Teile oder Aspekte von Programmen auch in Zukunft wieder zu verwenden. Das alte Paradigma der Software

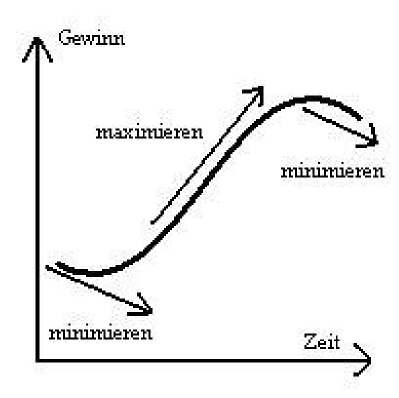
Entwicklung bedeutete, Software immer von Grund auf neu zu entwickeln. Programmierer waren teilweise sogar der Meinung, Programmcode müsse den gleichen urheberrechtlichen Regeln unterworfen sein wie literarische Werke. [12] Spätestens seit dem Aufkommen der objektorientierten Programmierung hat sich dieses Paradigma geändert und Programmcode wird nun als Werkzeug betrachtet und wenn sinnvoll wieder verwendet. Neben der technischen Ebene, auf der Reuse Zeitersparnis für den Programmierer bedeutet, kann auf der Anwendungsebene eine Wiederverwendung von Komponenten die Vertrautheit mit dem neuen Gesamtprodukt erhöhen. Abgewogen werden müssen hier der Verlust von Verbesserungsmöglichkeiten und die leichtere Erlernbarkeit des Programms.

Um das Verhalten eines Programms mit den Erwartungen der Benutzer zu harmonisieren, kann man, zusätzlich zum Gedankenmodell das der Benutzer vom Programm hat, ein **Verhaltensmodell** eines Systems erstellen. Ein einzelnes Verhalten ist optimalerweise eine einzelne Funktion, die eine für den Benutzer klar umrissene Aufgabe hat und deren Ergebnis eindeutig verifizierbar ist. Hilfreich um dieses Verhalten dem Benutzer deutlich zu machen ist der Einsatz von Geräuschen, die sofort Rückmeldung über den Erfolg geben. Die verschiedenen Verhaltensweisen eines Systems setzen sich zusammen aus den Erwartungen der Benutzer, Grenzen der Machbarkeit und Verhaltensmustern an die sich das System halten soll. Das Verhaltensmodell eines Systems stellt klar was das Programm tut, was es tun kann und was es nicht tun kann oder soll. Über Aktionsdiagramme wird das Verhaltensmodell spezifiziert. Im Laufe des Prozesses der Erstellung eines Verhaltensmodells werden sämtliche Erwartungen der Benutzer in das Verhalten des Systems integriert. Trotzdem auftauchende Unterschiede zwischen den Erwartungen der Benutzer beziehungsweise ihren Gedankenmodellen und dem Verhaltensmodell des Programms können dann ausgehend vom Verhaltensmodell behoben werden. Zur Anwendung kommen kann und soll es überall dort, wo es nötig ist, das Verhalten des Programms zu kennen, beispielhaft sind hier Dokumentation und Hilfestellungen. [14]

Partizipation von Benutzern in die Entwicklung eines Softwareprojekts ist heutzutage selbst im Falle von Produkten für anonyme Benutzer schwer wegzudenken, hier können beliebige unbefangene Personen als Tester dienen. Üblich ist das Einbeziehen der Benutzer in die Anforderungsanalyse und die Testphase, darüber hinaus ist ihre Beteiligung an allen anderen Phasen ebenfalls möglich und in einigen Modellen üblich. Ideal wäre es, sämtliche bekannte und potenzielle Benutzer einzubeziehen (teilweise im Internet üblich), realistischer ist die Einbeziehung von repräsentativen Benutzergruppen. Nicht immer sind einbezogene Benutzer freundlich gesinnt, Sicherheitsfirmen bedienen sich öfter verschiedener Angreifer, die sich zur Verfügung stellen müssen, um dem Gefängnis zu entgehen. Fragwürdig ist in diesem Fall ihre Glaubwürdigkeit, aber ein nicht krimineller Tester wäre aufgrund seiner unangemessenen Grundeinstellung kaum nützlicher. Alternativ werden oft Tester durch Belohnung stimuliert, ein kriminelles Verhalten an den Tag zu legen. Weiters können nicht nur freiwillige Benutzer am Projekt beteiligt werden, sondern auch unwissende oder unfreiwillige. Unerkannte Beobachtung kann in der Anforderungsanalyse nützlich sein und Testobjekte müssen nicht immer wissen, dass sie gestestet werden, siehe Sicherheitssysteme. [5] Partizipation, die über Anforderungsanalyse und Test hinausgeht, zielt in zwei Richtungen. Zum einen wird die Erfüllung der Erwartungen noch sicherer gestellt. Wenn Kunde und Benutzer in den gesamten Prozess eingebunden sind, ist eine Fehlinterpretation der Anforderungen sehr unwahrscheinlich und die Benutzer werden mehr oder weniger exakt das Produkt bekommen, das sie gewollt haben. Aber zum anderen hat eine derart enge Einbeziehung auch eine andere Wirkung. Nicht nur die Benutzer prägen dadurch das System, sondern auch das System die Benutzer. [13] Erstens hat eine derart enge Teilnahme eine sehr gute Lernwirkung, Eine spätere Notwendigkeit des vertraut Machens fällt weg. Und zweitens sind Mitarbeiter, die in die Entwicklung des Systems eingebunden wurden, zur Loyalität dem Programm gegenüber verpflichtet, da sie es selbst entwickelt haben. Wer ist schon bereit seine eigene Leistung herabzusetzen. So kann sichergestellt werden, dass Benutzer ein Programm später auch akzeptieren, selbst wenn es noch Fehler enthält.

Veränderungsmanagement ist eine Technik, die großteils auf praktischer Erfahrung von Managern beruht. Dabei geht es nicht darum, das Ziel einer Veränderung festzulegen oder zu bewerten, ob eine Veränderung richtig oder falsch ist. Veränderungsmanagement kann man in jeder Situation bei jeder Veränderung anwenden, es interessiert sich für den Prozess der Veränderung selbst. Dabei begründet es sich auf zwei grundlegende Ideen. Einerseits der S-Kurve, andererseits dem Konzept des „unfreezings". Die S-Kurve ist eine zweidimensionale Kurve, die horizontale Achse stellt hier die Zeit dar und die vertikale Achse steht für Leistung, Profit und Zufriedenheit. Nach der Einführung eines neuen Systems geht die Leistung anfangs etwas zurück, da

die Veränderung den Arbeitsablauf stört. Dann steigt die Kurve allmählich, wenn sich die Vorteile der Veränderung auswirken. Schließlich kommt es zu einer Stagnation oder sogar einem leichten Abfallen der Kurve, da sich die Benutzer an die neue Situation gewöhnen. An diesem Punkt ist die Entwicklung abgeschlossen und die Organisation wird voraussichtlich die nächste Veränderung in Angriff nehmen. Veränderungsmanagement versucht nun, diese Kurve zu optimieren. Der anfängliche Verlust soll minimiert werden, die Steigung soll so hoch hinauf wie möglich und der Verlust am Ende soll minimiert oder gar verhindert werden. [6]



Die zweite Sichtweise sagt, dass Mitarbeiter zuerst aus ihren Gewohnheiten gelöst werden müssen, indem man sie mit dem gegenwärtigen Zustand unzufrieden und rastlos macht (unfreeze), dann müssen sie nach vorne bewegt (move) und schließlich an den neuen Zustand gewöhnt werden (refreeze). Organisationen konzentrieren sich gewöhnlich auf den mittleren Schritt und als Folge kehren die Benutzer früher oder später in ihre alten Gewohnheiten zurück, da ihnen die Motivation zur Veränderung fehlt. Dies setzt veränderungsresistente Mitarbeiter voraus, eine pessimistische Sichtweise, die wohl genauso wenig zutrifft wie die vielfach geteilte optimistische Sichtweise, dass Mitarbeiter den Enthusiasmus ihrer Manager teilen. Die meisten Benutzer liegen irgendwo dazwischen und je nachdem wie veränderungsresistent sie sind ist diese Methode von großem Nutzen. Funktionieren soll dies, indem man den Mitarbeitern anfangs Gründe für die Veränderung gibt, das Gefühl, dass sie die Veränderung ausgelöst und in der Hand haben. Im Verlauf der Veränderung muss den Mitarbeitern klar sein, wieso sie genau den Weg gehen, den sie gehen. Und schlussendlich soll eine Stabilisierung durch Versorgung mit Erfolgserlebnissen folgen. [6]

Dokumentation ist als Teil der Benutzerschnittstelle zu betrachten und stellt oftmals den ersten Kontakt eines Benutzers mit dem System dar. Sie dient dazu, die intuitive Verständlichkeit eines Produkts zu ergänzen, wo dieses zu komplex ist, um sofort klar zu sein. Natürlich bleibt festzustellen, dass die Dokumentation alleine nicht ausreicht, eine Benutzerschnittstelle verständlich zu machen, und ein schlechtes Design der Benutzerschnittstelle kann dadurch nicht kompensiert werden, somit ist Dokumentation immer als ergänzend zu betrachten. Dokumentationen sollten sämtliche Funktionen und Aspekte eines Programms nachlesbar machen, speziell das Verhaltensmodell, das dem Produkt zugrunde liegt, sollte klar ersichtlich sein. Wiederholungen und unnötige Komplexität sind hingegen störend, wenn eine Dokumentation zu kompliziert ist, schließen die Benutzer daraus auf das Programm und werden abgeschreckt. Von den verschiedenen Formen der Dokumentation sind Tutorials für Anfänger am nützlichsten und Referenz Manuals für Experten von Interesse. User Guides sind vor allem für die durchschnittlich erfahrenen Benutzer interessant. Als Alternative zu schriftlichen Manuals stehen heutzutage meist online Hilfssysteme zur Verfügung, die die gleiche Funktion haben wie die Handbücher. Als eine Abart kann ein Hilfesystem so entwickelt werden, dass es das Verhalten des Benutzers überwacht und bei Fehlverhalten von selbst Hilfestellungen gibt. [7]

Schulungen sind die letzte hier erwähnte Methode um Benutzer mit einem System vertraut zu machen. Sie finden stets erst nach Fertigstellung des Produkts statt und können somit als teuer eingestuft werden. Sie finden dort Anwendung, wo Programme so kompliziert sind, dass sie nicht intuitiv verstanden werden und das Lernen per Dokumentation aufgrund der Komplexität zu ineffizient wäre.

5. Fazit:

Es ist in jedem Entwicklungsprozess einer Benutzerschnittstelle zu empfehlen, sämtliche Möglichkeiten zu nutzen, ein neues Produkt so zu entwickeln, dass es von den Benutzern später akzeptiert wird. Hier sind Faktoren wie kulturelle Prägung und bisherige Erfahrung sowie kognitive Fähigkeiten zu berücksichtigen. Die Möglichkeiten, hier eine hohe Akzeptanz zu erlangen, sind keine Wundermittel, aber ihre Einhaltung erleichtert die spätere Abnahme des Produkts. Spezieller Augenmerk ist auf Methoden zu legen, die vor oder während der Entwicklung eingesetzt werden können, sie stellen

sicher keine übermäßige Ressourcenbelastung dar und haben eine deutliche Wirkung.

6. Literatur:

[1] http://www.isys.uni-klu.ac.at/ISYS/Courses/03SS/eva/vounterlagen/9.10%20Gestaltung%20von%20Benutzungsschnittstellen

[2] Stefan Leuthold, „Benutzer sind vom Mars, Entwickler von der Venus“, http://www.stimmt.ch/knowledge/20030304/benutzervommars.pdf, 2003

[3] Joerg Kampmann, http://homepage.ruhr-uni-bochum.de/Joerg.Kampmann/HTML/ScreenDesign.htm

[4] Ernst Denert, „Software Engineering“, Springer Verlag, Heidelberg, 1991

[5] D.C. Gause, G.M. Weinberg, „Software Requirements – Anforderungen erkennen, verstehen und erfüllen“, Hanser Verlag, München, Wien, 1993

[6] Don Yeates, Maura Shields, David Helmy, “System Analysis and Design”, Pitman Publishing, London, 1994

[7] Deborah J. Mayhew, “Principles and Guidelines in Software User Interface Design”, Prentice Hall Verlag, New Jersey, 1992

[8] A. Russel Jones, “Linux vs. Windows: Choice vs. Usability”, http://www.devx.com/opensource/Article/16969

[9] Ashley George Taylor, “WIMP Interfaces”, CS6751 Topic Report: Winter '97, http://www.cc.gatech.edu/classes/cs6751_97_winter/Topics/dialog-wimp/

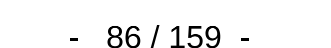
[10] “ETHICS – Backgrounds and ETHICS overview”, http://www.macs.hw.ac.uk/ism/ug2/Ass5/ethics.html,

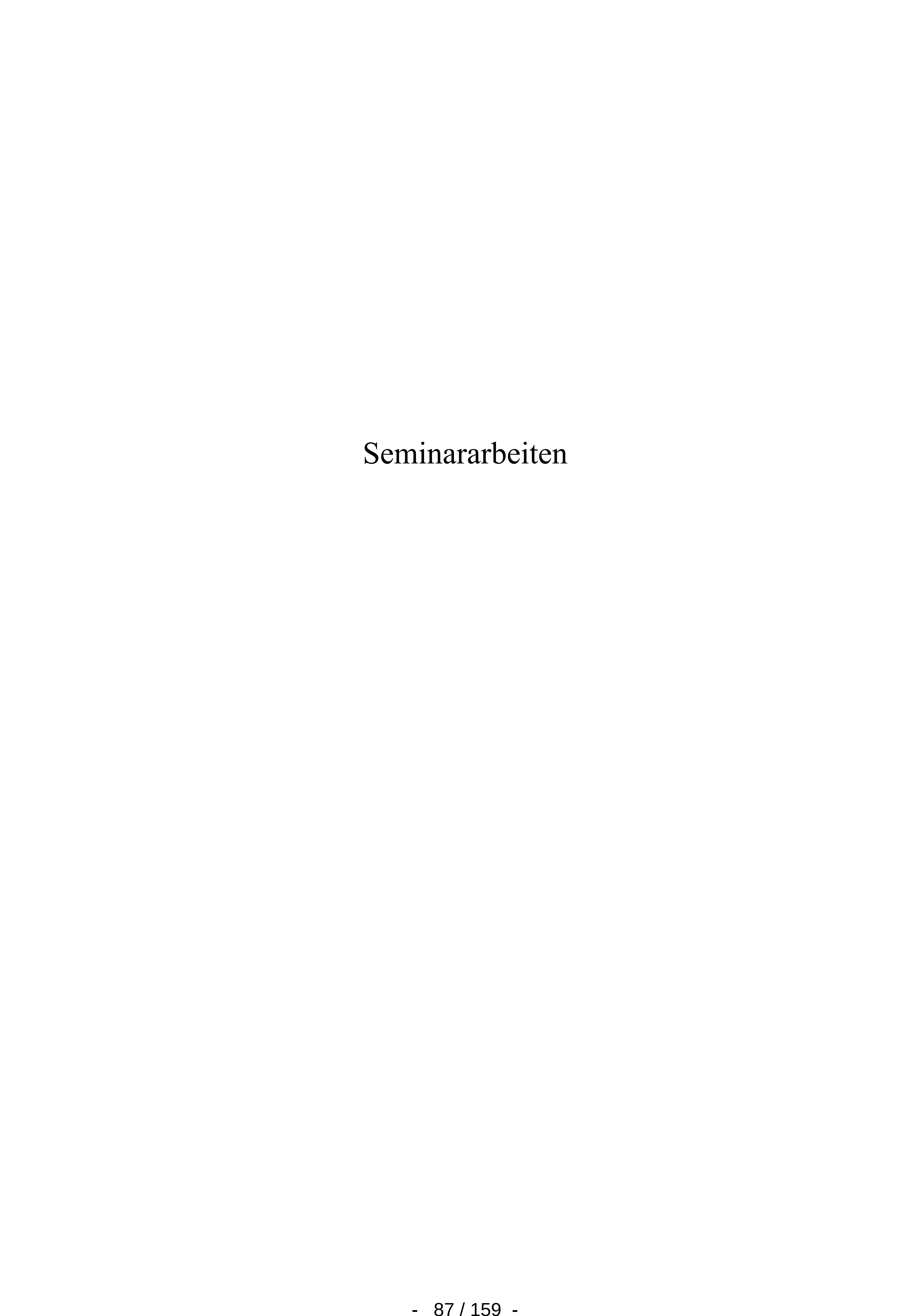
[11] Christian Dahme, Wolfgang Hesse, http://waste.informatik.hu-berlin.de/Dahme/edidahm.pdf, 1997

[12] Carma McClure, “The three Rs of Software Automation – Reengineering, Repository, Reusability”, New Jersey, 1992

[13] Enid Mumford, “Effective systems design and analysis requirements”, Macmillan Verlag, New York, 1995

[14] James A. Kowal, “Behavior Models – specifying user expectations”, Prentice Hall Publishing, New Jersey, 1992





Seminararbeiten

CMM – Eine Einführung

Gudrun Egger
9711612
gegger@edu.uni-klu.ac.at

Abstract

Die Ansprüche an die Softwareentwicklung wachsen ständig. Diesen Ansprüchen ist der Projektalltag oft nicht gewachsen. Die Gefährdung von Projekten ist aktueller denn je. Software-organisationen, die ihr Risiko erfolgreich managen, können jedoch gezielt entwickelt werden. Zertifizierungen zur Dokumentation von Qualitätsstandards gewinnen dadurch immer größere Bedeutung. So bietet das Capability Maturity Model (CMM), bzw. in der neuen Version das Capability Maturity Model Integrated (CMMI), gute Anhaltspunkte für die Gestaltung einer erfolgreichen Softwareorganisation. CMM konzentriert sich dabei auf die Prozesse, die an der Herstellung von Software beteiligt sind. Dahinter steht die Idee, dass die Qualität von Software wesentlich von den Entwicklungsprozessen der Hersteller abhängt.

1. Einleitung

Sehr wenige Projekte enden in Budget und Zeit mit der versprochenen Funktionalität. Um Qualität bei Softwareprojekten zu erreichen bietet das CMM Model einen Zertifizierungsstandard.

Das Capability Maturity Model für Software, entstanden am Software Engineering Institute (SEI) an der Carnegie Mellon University in Pittsburgh, Pennsylvania, wird weltweit angewendet, um die Art und Weise der Softwareherstellung und –Wartung zu verbessern. Das CMM hat seinen Ursprung in einem Auftrag des Department of Defence (DoD), das die mangelnde Qualität von Softwareprojekten in den Griff bekommen wollte. Unter der Leitung von Humphrey Watts begann das SEI im Auftrag vom DoD 1986 mit der Entwicklung eines Fragebogens, mit dessen Hilfe die Leistungsfähigkeit von Softwarelieferanten bewertet werden sollte. Fortwährende Überschreitungen von Lieferterminen, Entwicklungsbudgets sowie mangelhafte Qualität von Softwareprodukten veranlassten das DoD ein solches Hilfsmittel entwickeln zu lassen. Der Fragebogen wurde zu einem Referenzmodell ausgebaut, das als Vergleichsnorm für Softwarelieferanten dienen soll. Dieses Referenzmodell erhielt den Namen Capability Maturity Model (CMM). Die Version 1.0 wurde 1991 veröffentlicht. 1993 kam es auf Grund von Rückmeldungen aus der Industrie zu einer Überarbeitung des Modells, das zur verbesserten Version 1.1 führte [1]. 1997 / 1998 forderte das DoD eine Überarbeitung bzw. Umstrukturierung des Modells, welche zur Entwicklung des Capability Maturity Model Integrated (CMMI) führte.

Dieses Paper soll als eine Einführung in das Capability Maturity Model gesehen werden. Im Kapitel 2 werde ich mich mit Ursachen und Gründen für gefährdete Projekte befassen, da dies der Anlass dafür war, warum das SEI mit der Entwicklung des CMM begonnen hatte. Um die Qualität von Softwareprodukten zu garantieren, gilt das CMM für viele Firmen als ein Zertifizierungsstandard (Kapitel 3). Im Kapitel 4 werde ich genauer auf die Beschreibung der Reifegrade des Capability Maturity Model eingehen. Im Kapitel 5 erfolgt eine kurze Gegenüberstellung der CMM Modells mit der neuen Version CMMI. Im 6. Kapitel werden Erfahrungen von Organisationen, die CMM verwenden, kurz skizziert.

2. Ursachen und Gründe für gefährdete Projekte

Die Möglichkeiten der Softwareindustrie prägen die Visionen vieler Firmen. Weltweite, vernetzte Systeme, Verteilung mit Datenkonsistenz, E–Business Lösungen und vieles mehr sind Schlagworte, die Wirklichkeit werden sollen. Mit den Visionen wachsen auch die Ansprüche an die Softwareentwicklung. Ansprüche, denen der Projektalltag oftmals nicht gewachsen ist.

Projektabbrüche und Zeitüberschreitungen kosten meist viel Geld. Oft wird die gewünschte Funktionalität

nicht erreicht. Immer noch scheitern ca. 30% der Projekte, und im Schnitt werden Projekte doppelt so teuer wie veranschlagt. Dies kostet nicht nur Geld, sondern bedeutet auch verlorenen Chancen im Geschäft. In der Zwischenzeit sehen die Banken das Projektrisiko als ein Risiko an, das mit Eigenkapital zu hinterlegen ist. Erfolgreiche Softwareorganisationen managen effektiv ihr Softwarerisiko. Zeit, Kosten und Qualität werden so kontrolliert, dass Projekte im Zeit- und Kostenrahmen die gewünschte Funktionalität liefern. Dies minimiert das Projektrisiko. Darüber hinaus können Softwareorganisationen, die das Risiko ihrer Projekte effektiv managen, die vorhandenen Informationen zu einer Verbesserung innerhalb der Projekte und der Softwareorganisation nutzen.

Es gibt wohl kein Unternehmen, das keine erfolgreiche und effektive Softwareorganisation möchte. Dennoch erreichen in der Praxis nur wenige Unternehmen dieses Ziel. Erfolgreiche Softwareorganisationen müssen gezielt gestaltet werden.

3. Zertifizierungen zur Dokumentation von Qualitätsstandards

Um das Risiko der Projektgefährdung zu minimieren gewinnen Zertifizierungen aus einer Reihe von Gründen eine besondere Bedeutung. Zum einen sind die einer Zertifizierung zu Grunde liegenden Kriterien sehr gute Checklisten, um den eigenen „Reifegrad“ gezielt entwickeln und überprüfen zu können. Zum anderen können offizielle Zertifizierungen als Dokumentation eines Reifegrads nach außen (Kunden oder interne Auftraggeber) verwendet werden. Nicht zuletzt hat eine offizielle Zertifizierung aber auch den Effekt, dass die für einen Reifegrad notwendigen Maßnahmen (z.B. Kostenkontrolle) tatsächlich eingeführt werden, wenn sie für einen externen Auditor unabhängig überprüft werden.

Neben den ISO9000 gewinnt das amerikanische Capability Maturity Model (CMM), der Standard des SEI weltweit eine immer stärkere Bedeutung. CMM wird oftmals als Zertifizierungsstandard verwendet (“Unsere Organisation hat CMM Level 3”), es kann aber ebenso als Checkliste verwendet werden, um herauszufinden, was eine erfolgreiche und effektive Softwareorganisation auszeichnet.

4. CMM

CMM ist ein Modell, das beschreibt, wie sich Praktiken des Software Engineering in Organisationen unter bestimmten Bedingungen entwickeln:

- Die Arbeitsschritte werden als Prozess organisiert und betrachtet.
- Die Entwicklung des Prozesses wird systematisch geleitet [2]

Der Softwareprozess ist eine Menge von Werkzeugen, Methoden und Praktiken, die wir benötigen um ein Softwareprodukt herzustellen. Um den Fortschritt und die generelle Ausführungszeit des Softwareprozesses vorbestimmen zu können, ist es notwendig, dass die Teilbereiche des Softwareprozesses statistisch messbar sind. Das bedeutet, dass jeder Teilbereich bei jedem Durchlauf des Softwareprozesses innerhalb einer vorgegebenen statistischen Grenze dieselbe Ausführungszeit hat und dieselben Ergebnisse liefert. Erst unter Voraussetzung dieser statistischen Messbarkeit der Prozesse ist es möglich, eine kontinuierliche Verbesserung des Softwareprozesses anzustreben [5].

Das Capability Maturity Model definiert fünf Stufen der “Reife” einer Organisation bezüglich seiner Fähigkeiten, Software - Entwicklungsprojekte durchzuführen. Das Modell soll helfen, den Reifegrad von Softwareentwicklungsprozessen zu ermitteln, um gezielte Prozessverbesserungen durchführen zu können. Mit steigendem Reifegrad (maturity level) wird die Erwartung verbunden, dass die Vorhersagbarkeit von Terminen, Kosten und Qualitätszielen zunimmt. Für jede der fünf Stufen definiert das CMM, wie sich ein Prozess auf dieser Stufe darstellt. Die einzelnen Stufen bauen aufeinander auf. Eine Stufe setzt voraus, dass die Anforderungen an die Prozesse, die die anderen Stufen erfordern, erfüllt sind [1].

Jeder Reifegrad setzt sich jeweils aus einer Sammlung von Praktiken zusammen. In Organisationen, die einen bestimmten Reifegrad erreicht haben, sind diese Praktiken ein Teil der Routine und werden effektiv in den Prozess miteinbezogen. Die für den Reifegrad spezifischen Sammlungen von Software- und Management – Praktiken werden als Schlüsselprozessbereiche (SPBs) bezeichnet. Jeder SPB setzt sich aus Schlüsselpraktiken zusammen, die angeben, was zu tun ist, um den jeweiligen SPB zu erfüllen. Es wird jedoch nicht spezifiziert, wie genau dies zu tun ist [2].

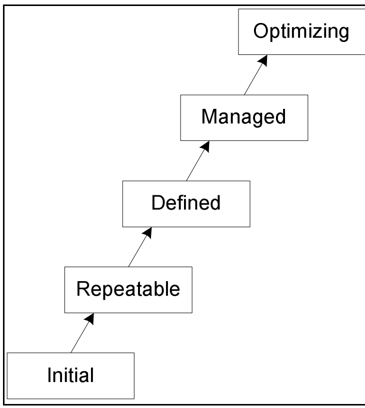


Abb. Reifegrade eines Prozesses

4.1. Stufe 1: Initial

Ein Prozess der Stufe 1 ist ad hoc und wird nur wenig gelenkt. Es gibt keine Vorgaben zur Planung und Steuerung von Projekten. Der Erfolg oder Misserfolg eines Projektes hängt in erster Linie von den Bemühungen, der Motivation und der Qualifikation der beteiligten Personen ab. Auf dieser Stufe arbeitet eine Organisation ohne effektive Kostenschätzung, Qualitätssicherung oder Projektplan. Ebenso wenig wird auf eine effiziente Zeitabschätzung der Entwicklung eines Produkts Wert gelegt. Weiters existieren keine einheitlichen Richtlinien um den Änderungen des Softwareprozess zu überwachen (Change control) [5]. Jedes Projekt einer Organisation der Stufe 1 wird als ein neues Projekt empfunden, da auf bereits gemachte Erfahrungen nicht zurückgegriffen werden kann.

Der Softwareprozess der Stufe 1 ist recht weit verbreitet: 32,2% der Software – Entwicklungsorganisationen, die dem SEI ihre Daten übermitteln, befinden sich auf Stufe 1. SPBs spielen auf Stufe 1 noch keine Rolle [2]

Sich auf Stufe 1 befinden, bedeutet nicht, dass eine Organisation nicht in der Lage ist, gute Software herzustellen. Aber es bedeutet, dass durch verpasste Deadlines und viele weitere Prozessunzulänglichkeiten die Kosten für Hersteller und Kunden zu hoch werden können.

Um zu dem nächsten Reifegrad zu gelangen, ist es notwendig ein Projektmanagement, eine Qualitätssicherung und ein Change Control Management einzuführen:

- Projektmanagement: Der Ablauf von Projekten muss detailliert geplant und eine genaue Zuteilung von Ressourcen zu den einzelnen Projektschritten vorgenommen werden. Für die erfolgreiche Projektdurchführung ist eine Einbindung des Managements unerlässlich. Projektpläne müssen vom Management reviewed und offiziell bewilligt werden.
- Die Qualitätssicherungsgruppe stellt sicher, dass es zu keinen Abweichungen von definierten Richtlinien kommt. Sie ist unmittelbar der obersten Führungsebene untergeordnet und erstellt für diese laufende Reviewberichte.
- Die Change Control befasst sich mit der Kontrolle und Überwachung von Änderungen, die sich aus den einzelnen Prozessschritten ergeben. Änderungen entstehen in jeder Prozessphase: Anforderungsanalyse, Design, Implementierung, Test. Für jede Änderung muss festgehalten werden, welche Auswirkungen sie auf andere Prozessschritte hat, welche Personen davon betroffen sind und von wem sie bewilligt werden muss. [5]

4.2. Stufe 2: Repeatable

Auf Stufe 2 gibt es strukturierte Anforderungen an den Prozess [1]. Der Prozess untersteht einer Prozessdisziplin, so dass erfolgreiche Projekte in Bezug auf Kosten, Zeitplan und Anforderungen die Norm darstellen. Projektleiter sind in der Lage, angemessene Schätzungen und Projektpläne zu realisieren. Anhand dieser Pläne können sie die Projektdurchführung verfolgen und lenken [2]. Organisationen der Stufe 2 nützen Erfahrungswerte aus abgeschlossenen Projekten und können ähnliche Softwareprozesse effizient durchführen. Sie haben jedoch Umstellungsprobleme, wenn sie mit gänzlich neuen Prozessen konfrontiert werden [5].
Stufe 2 sind folgende SPBs zugeordnet:

Das **Anforderungsmanagement** und damit die Lenkung der Anforderungen stellen einen entscheidenden Faktor bei der Stabilisierung des Softwareprozesses von Stufe 1 dar. Nur ein stabiler Prozess macht Erfolge wiederholbar, nicht gelenkte Anforderungen führen zu verspäteten Produktlieferungen und schlechter Qualität.

Die **Softwareprojektplanung** geht auf eine Reihe von Problemen bei Softwareprojekten ein. Ziele dieses SPB sind:

- Softwareschätzungen in Bezug auf Umfang, Zeitplan, Aufwand, etc. werden dokumentiert, um sie für die Planung und Verfolgung des Softwareprojekts zu nutzen.

- Die Aktivitäten und Kompetenzen des Softwareprojekts werden geplant und dokumentiert.
- Die beteiligten Personen und Gruppen vereinbaren ihre Kompetenzen in Bezug auf das Softwareprojekt.

Die Aktivitäten des Prozessbereichs **Softwareprojektlenkung und –Verfolgung** ermöglichen einen Einblick in die Aktivitäten und den Status eines Projekts. Weiterhin ermöglichen sie, die Projektaktivitäten zu überwachen und Lenkungsmaßnahmen zu ergreifen. Sie nutzen den Softwareentwicklungsplan des Projekts als Basis für Überwachung und Lenkung.

Das **Software – Unterauftragnehmermanagement** befasst sich mit der Auswahl und dem Management von Komponentenlieferanten für das Softwareprojekt.

Die **Softwarequalitätssicherung** sorgt dafür, dass das Projekt seinem Prozess treu bleibt und verschafft dem Management Einblick in den Prozess. Sie signalisiert dem Management, wenn die Lenkungsmaßnahmen der Softwareprojektlenkung und –Verfolgung unzureichend sind.

Das **Software – Konfigurationsmanagement** bezieht sich auf die Lenkung sowohl des Produkts, das zur Lieferung bestimmt ist, als auch der Zwischenprodukte, die im Laufe des Projekts entstehen [2][3].

Um den nächsten Reifegrad zu erreichen ist die Gründung einer Prozessgruppe und die Einführung einer Software – Entwicklungsprozess – Architektur (SEA) notwendig. Die Aufgabe der Prozessgruppe ist es Verbesserungen am Softwareprozess vorzunehmen. Sie erstellt periodisch Reviews über den Projektstatus und –fortschritt und identifiziert neue für den Prozess relevante Technologien. In der SEA wird der Softwareprozess in kleine definierte Einheiten (Tasks) unterteilt, wobei für jeden Task dessen Vorbedingungen, Funktionalität und ausführende Organisationseinheit dokumentiert werden [5].

4.3. Stufe 3: Defined

Auf Stufe 3 sind die wesentlichen Voraussetzungen für einen erfolgreichen Softwareprozess bekannt und werden organisationsweit erfüllt. Der Prozess der Organisation ist als Standard definiert und in sich konsistent [1]. Individuelle Prozessaktivitäten sind sichtbar, da eine SEA eingeführt wurde. Alle Projekte basieren auf demselben Standardprozess, wobei jede Abweichung von diesem bewilligt und dokumentiert werden muss. Zusätzlich wird ein Risikomanagement eingeführt, das sich damit beschäftigt, Risikobereiche innerhalb des Softwareentwicklungsprozess zu identifizieren und geeignete Maßnahmenpläne auszuarbeiten [5].

Folgende SPBs sind Stufe 3 zugeordnet:

Der **Organisationsweite Prozessfokus** befasst sich, wie die **Organisationsweite Prozessdefinition**, mit dem Sammeln und Übertragen der Prozessverbesserungen und besten Praktiken auf andere Projekte und die gesamte Organisation.

Das **Trainingsprogramm** ist ein SPB der Stufe 3, doch führen Organisationen auf allen Reifegraden Trainings durch. Spätestens auf Stufe 3 sollte aber ein effektives, organisationsweites Trainingsprogramm vorhanden sein.

Auf Stufe 2 befassten sich die Softwareprojektplanung und die Softwareprojekt-lenkung und –Verfolgung sowohl mit den technischen als auch mit den programmatischen Aspekten eines Softwareprojekts. Auf Stufe 3 stellt das **Integrierte Softwaremanagement** eine Weiterentwicklung dieser Praktiken an, indem zunächst der Softwareprozess der Organisation berücksichtigt und dann an Projekte angepasst wird.

Das **Softwareprodukt Engineering** befasst sich mit der Anforderungsanalyse, Design, Code und Test. In diesem SPB werden ganz konkret das Engineering und die Produktion der Software durchgeführt. Das bedeutet jedoch nicht, dass sich die anderen SPBs nicht mit dem eigentlichen Software Engineering beschäftigen. Stattdessen bedeutet es, dass die meisten Hindernisse für die konsistente und effektive Produktion von Software sich daraus ergeben, wie die Arbeit organisiert ist. Und genau dies wird in den anderen SPBs thematisiert.

Der Zweck der **Gruppenkoordination** ist, einen Mechanismus zu schaffen, durch den die Software Engineering Gruppen sich bei den anderen Engineering Gruppen beteiligen können. Als Resultat sollten alle Bedürfnisse des Kunden in Bezug auf das gesamte Projekt, einschließlich der Software, befriedigt werden.

Peer Reviews sollen effektiv verhindern, dass Fehler bis zu späteren Schritten im Life Cycle unbemerkt bleiben. Der Begriff „Peer Review“ umfasst jede systematische Untersuchung eines Arbeitsprodukts durch einen Kollegen bis hin zu einer sehr formellen Software – Inspektion [2][3].

Für die Erreichung des nächsten Reifegrades werden Maßzahlen eingeführt, die Aufschluss über die Effizienz des Prozesses geben. Für jeden Prozessschritt muss festgelegt werden, welche Kosten durch ihn entstehen bzw. wie dessen Qualität gemessen werden kann. Zu diesem Zweck müssen vorab Kennzahlen für die Qualitätsbewertung definiert werden. Weiters

werden die Kosten für die Korrektur eines fehlerhaften Prozessschrittes spezifiziert [5].

4.4. Stufe 4: Managed

Eine Organisation der Stufe 4 steuert die Softwareentwicklung. Auf dieser Stufe werden die Qualität der Produkte und die Produktivität der Prozesse durch ein organisationsweites Metrikprogramm quantitativ gemessen. Dies dient zur vorausschauenden Projekt- und Organisations-steuerung. Durch die umfassenden Prozessmessungen und die darauf aufbauenden Analysen können die Produktqualität sowie der Entwicklungsprozess kontinuierlich verbessert werden. Für die Organisation ist der Fortschritt zu Reifegrad 4 mit hohen Kosten für die Metrikerfassung und –Verwaltung verbunden. Um die Gültigkeit der Metriken sicherzustellen, müssen einheitliche Kriterien für die Datenerfassung festgelegt werden, damit den Bewertungen und Analysen keine invaliden Daten zugrunde liegen [5].
Es gibt nur zwei SPBs der Stufe 4:

Das **Quantitative Prozessmanagement** betrifft die Prozessqualität und die Messung der Prozessleistung. Es dient dazu besondere Ursachen von Abweichungen im Softwareprozess zu beseitigen. Eine besondere Abweichung ist ein vorübergehender Umstand, der eine unerwartete Abweichung in der Prozessleistung hervorruft. Dieser SPB verfolgt die folgenden Ziele:

- Die Praktiken des Prozessmanagements erfolgen nach einem zuvor erstellten Plan
- Die Prozessleistung des projektdefinierten Softwareprozesses wird quantitativ gelenkt.
- Die Prozessfähigkeit des Softwareprozesses der Organisation ist in ihrer Quantität bekannt.

Software – Qualitätsmanagement befasst sich mit der Produktqualität. Folgende Ziele sollen erreicht werden:

- Die Projektaktivitäten für die Softwarequalität werden geplant. Dieser SPB ist Teil des Softwareproduktplans, der durch das Integrierte Softwaremanagement (SPB Stufe 3) vom projektdefinierten Softwareprozess abgeleitet wird.
- Messbare Ziele für Softwareproduktqualität, die mit dem Plan erreicht werden sollen, und deren Einstufung werden definiert.
- Der konkrete Fortschritt zum Erreichen der Qualitätsziele für die Softwareprodukte wird quantifiziert und geleitet.

Diese Ziele sollen sich in zwei Ergebnissen für die Organisation widerspiegeln: Dem quantitativen Verständnis der Qualität von Softwareprodukten und dem Erlangen einer höheren Produktqualität [2][3].

Zur Erreichung des nächsten Reifegrades wird ein System zur automatischen Erfassung von Prozessdaten eingeführt. Dadurch kann auf ein umfassenderes Datenmaterial zugegriffen werden, das für die Prozessanalyse und die Prozessverbesserung herangezogen werden kann [5].

4.5. Stufe 5: Optimizing

Auf Stufe 5 ist der zuverlässige Ablauf des Softwareprozesses Routine und ermöglicht es den Beteiligten, sich auf eine kontinuierliche Prozessverbesserung zu konzentrieren. Alle Mitarbeiter wissen, welche Aufgaben wann zu erledigen sind. Sie konzentrieren sich nun darauf, den Softwareprozess so zu verändern um Wettbewerbsvorteile durch höhere Qualität und Produktivität zu erzielen. Sobald die Produktionsmechanismen effektiv und effizient sind, sowie gelenkt werden, können sich die Mitarbeiter auf deren Verbesserung konzentrieren [2].

Im optimierten Prozess wird großes Augenmerk darauf gelegt, Fehler innerhalb des Prozesses möglichst früh zu erkennen und die Kosten für deren Beseitigung möglichst gering zu halten. Dies kann u. a. durch vermehrte Inspektionen erreicht werden. Zusätzlich wird die Qualität des Prozesses dadurch verbessert, dass umfangreiche Prozessdaten fortlaufend erfasst und ausgewertet werden [5].
SPBs auf dieser Stufe sind:

Die **Fehlervermeidung** zielt darauf ab, allgemeine Ursachen für Prozessabweichungen zu lenken. Dass dieser SPB auf Stufe 5 platziert ist, bedeutet jedoch nicht, dass Organisationen auf niederen Reifegraden keine Fehlervermeidung praktizieren. Eine stabile Form der Fehlervermeidung basiert auf folgenden Aktionen: Analyse der Vergangenheit, Vorhersage des Entwicklungstrends, Erkennen der grundlegenden Ursachen und Ergreifen von präventiven Maßnahmen.

Technologie – Change – Management, **Prozess – Change - Management**: Die Organisation stellt fest, dass Wettbewerbsvorteile und finanzielle Stabilität durch eine systematische und kontinuierliche Prozessverbesserung erzielt werden können. Sie kommt zu der Erkenntnis, dass sie ihre Softwarequalität ausbauen kann, indem sie den Softwareprozess ständig weiterentwickelt. Diese Erkenntnis, wie Veränderungen herbeigeführt werden können, spiegelt sich in den oben genannten SPBs wider. Dieser SPB sollten in einer besseren Qualität und Produktivität resultieren [2][3].

4. CMMI – Vergleich zu CMM

CMM Integration ist die neue, überarbeitete Version vom Capability Maturity Model. CMMI hat mehr und detailliertere Hinweise zur Umsetzung der Prozesse der einzelnen Stufen. CMMI enthält somit mehr Informationen, aber nicht unbedingt mehr Anforderungen als CMM. In CMMI wurden die Prozesse der einzelnen Stufen neu arrangiert. Der Anwendungsbereich ist über die Softwareentwicklung hinaus zur Systementwicklung und Kauf von Software erweitert worden.

CMMI definiert neben dem stufenförmigen Modell mit den bekannten fünf Reifegraden auch ein kontinuierliches Modell, indem ein Unternehmen unabhängig von Reifegraden bewertet werden kann. Bei der kontinuierlichen Darstellung ist eine feinere, themenbezogene Darstellung der Reife einer Organisation möglich, indem es pro Schlüsselprozessbereich je einen Fähigkeitsgrad auf einer Skala von 0 bis 5 gibt. Fähigkeitsgrade beziehen sich somit auf einen Schlüsselprozessbereich, Reifegrade hingegen beziehen sich auf die Gesamtheit der Schlüsselprozessbereiche einer Stufe.

5. CMM in der Praxis

Das SEI publizierte 1996 einen Erfahrungsbericht von Unternehmen, die CMM in einem Zeitraum von ein bis drei Jahren zur Prozessverbesserung eingesetzt haben. Von den 138 befragen Organisationen befanden sich nach Selbsteinschätzung 87 auf Level 1, 36 auf Level 2 und 15 auf Level 3 oder höher. Die Befragung hatte das Ziel die CMM Reifegrade hinsichtlich folgender Kriterien zu untersuchen: Einhalten von Termin- und Budgetplänen, Kundenzufriedenheit, Produktqualität, Arbeitsmoral und Arbeitsproduktivität. Bei folgenden vier Kriterien konnte ein statistisch relevanter Zusammenhang zwischen höherem Reifegrad und Leistungsfähigkeit in einem bestimmten Bereich festgestellt werden: Einhalten von Terminplänen, Produktqualität, Arbeitsmoral, Arbeitsproduktivität. Die Kundenzufriedenheit bei Unternehmen auf Level 2 war tendenziell niedriger als bei Organisationen der Stufe 1, jedoch nahm die Zufriedenheit bei Level 3 wieder sehr stark zu.

Weiters wurden die Unternehmen befragt, inwieweit sich ihre Softwareprozesse durch die Einführung von CMM verbessert haben. Mehr als die Hälfte der Befragten gab an, dass sich der Prozess gut bis sehr stark verbessert habe. Immerhin 30% waren der Meinung, dass es eine teilweise Verbesserung gab und 14% sprachen von geringem bis keinem Erfolg.

Unternehmen, die ihre Softwareprozesse mit CMM verbessern konnten, führten folgende Gründe für die erfolgreiche Umsetzung an:

- Das Management überwachte aktiv den Fortschritt der Prozessverbesserung.
- Es gab klar definierte, verständliche Ziele der Prozessverbesserung, sowie genaue Vorgaben bezüglich der Kompetenzzuteilung.
- Das technische Personal wurde stark in die Prozessverbesserung miteinbezogen.
- Es gab ausreichend Ressourcen, die der Prozessverbesserung zugeteilt wurden.

Organisationen, denen es nicht gelungen ist, ihre Prozesse mit CMM zu verbessern, nannten folgende Merkmale:

- Obwohl es den Unternehmen bewusst war, welche Veränderungen sie in ihren Prozessen vorzunehmen haben, hätten sie mehr Hilfestellungen dafür benötigt wie sie die Maßnahmen umsetzen können.
- Die Softwarefirmen sehen in dem Verbesserungsprozess eine zusätzliche Belastung, die sie von der eigentlichen Arbeit abhält.
- Sie sind entmutigt durch schlechte Erfahrungen mit zuvor gescheiterten Prozessänderungsversuchen. [4]

6. Conclusio

CMM zeigt auf, was eine erfolgreiche Softwareorganisation auszeichnet: nämlich die Durchführung von Änderungen. Das bedeutet, dass die Organisationen, die dem CMM folgen, ganz gleich auf welcher Stufe sie sich befinden, zumindest eine Gemeinsamkeit haben: Sie alle konzentrieren sich auf Prozessverbesserungen. Durch die kontinuierliche Entwicklung ihres Prozesses steigt die Organisation von Stufe 1 auf Stufe 2, von da auf Stufe 3 etc.

Es bedeutet weiterhin, dass Softwareprojekte nicht nur Aktivitäten umfassen, die das Entwickeln von Software enthalten. Wenn der Kunde bei einem Produkt Zuverlässigkeit und Wiederholbarkeit erwartet, sollte man den gesamten Produktionsprozess betrachten, ganz gleich, ob es sich um die Entwicklung oder die Wartung von Software handelt.

7. Literaturverzeichnis

[1] Helmut Balzert, *Lehrbuch der Software – Technik*. Software – Management, Software – Qualitätssicherung, Unternehmensmodellierung, Akad. Verlag, Heidelberg [u.a.], 1998.

[2] Kenneth M. Dymond, *CMM® Handbuch*. Das Capability Maturity Model® für Software, Springer – Verlag, Berlin [u.a.], 2001.

[3] James D. Herbsleb, David Zubrow, Dennis R. Goldenson, Will Hayes, Mark Paulk, "Software Quality and the Capability Maturity Model", *Commun. ACM, 40, 6,* ACM Press, 1997, pp. 30-40.

[4] James D. Herbsleb, Dennis R. Goldenson, „A systematic survey of CMM experience und results", *Proceedings of the 18th international conference on Software engineering*, IEEE Computer Society, Berlin, 1996, pp. 323-330.

[5] Watts S. Humphrey, *Managing the Software Process*, Addison-Wesley, Boston [u. a.] 1989

Entwicklung von ISO 9000

Birgit Antonitsch (0060646), Hubert Gressl (0060188)
{bantonit, hgressl}@edu.uni-klu.ac.at
Diplomstudium Informatik

Abstract

Die DIN EN ISO 9000 ist eine Norm für Qualitätsmanagement und hat sich seit der Einführung in der Wirtschaft weit verbreitet. In vielen Branchen ist eine Zertifizierung dieser Art bereits zu einem „muss“ geworden.

Diese Arbeit behandelt die Entwicklung und den Aufbau der ISO 9000 Normfamilie, wobei die Veränderungen, die diese Norm erlebt hat, gezeigt werden. Im Zuge dessen wird auf die grundlegenden Prinzipien der einzelnen Versionen eingegangen. Die aktuell gültige Norm ist die ISO 9000 Version 2000, die genauer betrachtet wird. Zum Schluss wird die ISO 9000 Version 2000 mit anderen Qualitätsmanagementsystemen verglichen und die Eignung in Bezug auf die Softwareentwicklung behandelt.

1 Einleitung

Die DIN EN ISO 9000 ff. ist eine Normfamilie um Qualitätsmanagementsysteme international zu vereinheitlichen. Das Qualitätsmanagement, kurz QM genannt, soll sicherstellen, dass Güter, Dienstleistungen, Software und Prozesse den Anforderungen entsprechend hergestellt werden. Die Hauptaufgaben sind eine umfassende Qualitätspolitik und die Entwicklung von QM-Zielen, sowie die Verantwortungen für die Planung, Durchführung und Kontrolle dieser Ziele festzulegen. Die Verantwortung dieser Aufgaben liegt bei der Unternehmensführung, welche die Normfamilie als wirkungsvolles Hilfsmittel zur ständigen Qualitätsverbesserung einsetzen soll, um den Anforderungen am Markt gerecht zu werden. Weiters muss der gesamte betriebliche Ablauf dokumentiert werden. Eine Zentrale Rolle dabei übernimmt das QM-Handbuch (QMH), in dem die Zuständigkeiten und Regeln nach denen die Geschäftsprozesse ablaufen niedergeschrieben werden.

Unternehmen können sich ihr QM-System mit einem Zertifikat bestätigen lassen. Diese Zertifizierung erfolgt nach entsprechenden Normen, z.B. der ISO 9000 ff., und wird von unabhängigen Zertifizierungsstellen durchgeführt.

Dabei ist wichtig anzumerken, dass ein Zertifikat eines Unternehmen nichts über die Güte des Produktes, der Dienstleistung oder der Software aussagt, sondern zeigt, dass die Unternehmensleistungen durch definierte und dokumentierte Prozesse erstellt worden sind. Es ist möglich, dass ein Produkt eines nicht zertifizierten Unternehmens die gleiche oder sogar höhere Qualität als ein ordnungsgemäß zertifiziertes Unternehmen aufweist. Diese unzertifizierten Unternehmen werden ihre Leistungen ebenfalls über beschreibbare und auch reproduzierbare Prozesse erstellen, nur eben ohne Zertifikat. Aber dass ein chaotisch funktionierendes Unternehmen Produkte mit hoher Qualität herstellt, geschieht eher selten.

Die Zahl der zertifizierten Unternehmen hat in den letzten Jahren stark zugenommen. Im folgenden Absatz möchten wir auf die Gründe dieser Entwicklung eingehen.

1.1 Vorteile einer Zertifizierung

Die Zertifizierung ergibt folgende Vorteile:

- Werbung: Mit der Zertifizierung gewinnt man das Vertrauen der Kunden und Lieferanten.
- Reduziert das Produkthaftungsrisiko: Die Gesetzgebung besagt, dass der Hersteller seine Unschuld am Produktfehler beweisen muss. Ein Zertifikat über ein vorhandenes QM-Systems und eine genaue Produktdokumentation könnte die Einhaltung der Sorgfaltspflicht belegen.
- Die Auftragsbeschaffung wird erleichtert, da viele Auftraggeber eine anerkannte Zertifizierung eines QM-Systems fordern.
- Optimierung der Arbeitsabläufe: Durch die Vorbereitung auf die Zertifizierung, werden Arbeitsabläufe im Unternehmen genau analysiert und können im Zuge dessen optimiert werden.
- Die Zertifizierung fordert die Definition von wichtigen Dokumenten und die Regelung von Zuständigkeiten und Befugnissen. Dadurch wird das Qualitätsbewusstsein der Mitarbeiter und der Geschäftsführer erhöht.

Die Einführung eines QM-Systems in ein Unternehmen ist sehr wichtig und aufgrund der oben

genannten Argumente macht es auch Sinn dieses zertifizieren zu lassen. Warum die ISO 9000 ff. eine attraktive Variante ist, wollen wir im Weiteren behandeln.

2 Geschichte der Normfamilie ISO 9000

Um eine Grundlage für die Geschichte der ISO 9000 Normfamilie zu schaffen, wird der Begriff Norm erklärt und die Gremien, die für diese Normen verantwortlich sind, vorgestellt.

2.1 Norm

Der Begriff Norm laut DIN 820 Teil 1:

> „Normung ist die planmäßige, durch die interessierten Kreise gemeinschaftlich durchgeführte Vereinheitlichung von materiellen und immateriellen Gegenständen zum Nutzen der Allgemeinheit.“ [Schö01, S. 8]

Die Aufgabe einer Norm ist ein einheitliches Verständnis von Begriffen und Abläufen zu schaffen, um die Kommunikation zwischen zwei Partnern zu erleichtern. Der Handel sollte dadurch effizienter, sicherer und gerechter gestaltet werden. Ebenso sollten die Normen für die Kunden und Benutzer ein Hilfsmittel für die Überwachung von Produkten und Dienstleistungen sein. Genormte Produkte sollen definierte Eigenschaften aufweisen und mit einem, nach der selben Norm gefertigten, Produkt eines anderen Herstellers ausgetauscht werden können.

2.2 Normungsinstitute

2.2.1 ISO - International Organization for Standardization.

Diese internationale Organisation wurde am 26. Oktober 1946 gegründet und ihr Hauptquartier liegt in Genf. Im Mai 2004 betrug die Anzahl der beteiligten Länder 148. Die Aufgabe der ISO ist die Erarbeitung von internationalen Normen, um die unterschiedlichen nationalen Normen anzugleichen und zu vereinheitlichen. Sie sind eine Grundlage für den freien Welthandel. Die Erarbeitung der Normen wird in Technische Komitees (Technical Committees, kurz TCs genannt) erledigt. Im Mai 2004 existieren 225 solcher TCs. Die Gesamtanzahl der von der ISO verabschiedeten Normen seit ihrer Gründung beträgt ca. 13.700. [vgl. www.iso.org]

2.2.2 CEN - Comité Européen de Normalisation

Das Europäische Komitee für Normung wurde 1961 in Paris gegründet. 1975 wurde der Hauptsitz, das CEN Management Center, nach Brüssel verlegt und bekam den Status eines eingetragenen Vereines nach belgischem Recht. Im Dezember 2003 betrug die Anzahle der aktiven Komitees 281. Die Zahl, der von diesen Komitees erstellten Normen, betrug 6.772 im Dezember 2003. Das Ziel dieses Gremiums ist die Schaffung eines einheitlichen und modernen Normenwerkes für Europa. Es besteht aber eine starke Zusammenarbeit mit der ISO, da nur eine weltweit einheitliche Norm die gewünschten Vorteile bringt. [vgl. www.cenorm.be]

2.2.3 DIN - Deutsches Institut für Normung

Dieses Institut wurde 1917 gegründet und hat seinen Sitz in Berlin. Es ist ein Verein nach deutschem Recht. Auch die nationalen Normungsinstitutionen pflegen eine enge Kooperation mit den internationalen Normungsgremien. Viele Normen werden so institutionsübergreifend erstellt und bearbeitet. Bei der DIN wird die Arbeit von 76 Normungsausschüssen, die sich weiter in 3.306 Arbeitsausschüsse unterteilen, erledigt. Diese Ausschüsse haben seit der Gründung bereits 28.069 Normen veröffentlicht. [vgl. www.din.de]

2.3 Entwicklung

Zu Beginn der sechziger Jahre entstanden in großen Unternehmen eigene Abteilungen zur Qualitätssicherung. Deshalb begann parallel dazu die Arbeit an der Entwicklung von Standards und Normen für Qualitätsmanagement. Die ersten Standards entstanden in Bereichen des Militärs, der Nuklearindustrie und der Automobilindustrie.

Um diese Standards aufgrund der Anforderungen der Wirtschaft, deren unterschiedlichen Branchen und Interessensgruppen zu vereinheitlichen, setzte die ISO 1979 ein Technisches Komitee ein. Der Titel dieses TC 176 war *„Quality management and quality assurance“* und ihr Ziel war es einen Standard zu entwickeln, der QM-Systeme international vereinheitlicht. [vgl. ISO094, S.2]

1985 erschienen die ersten Entwürfe der ISO 9000, 9001, 9002, 9003 und 9004. Diese Entwürfe wurden von den verschiedenen Mitgliedsländern der ISO bearbeitet und die endgültige Fassung der Normfamilie wurde 1987 von der ISO verabschiedet. Damit die Länder der Europäischen Union konkurrenzfähig gegenüber den USA und Japan blieben, wurde kurz darauf durch das CEN eine englische Version als EN 9000:1987 ff herausgegeben. Da alle Mitgliedsstaaten verpflichtet waren diese Normreihe anzuerkennen, wurden andere Normen verdrängt und es kam zu einer Weiterverbreitung des ISO 9000 Standards.

1990 wurde die Normfamilie DIN EN ISO 9000 bzw. ÖNORM EN ISO 9000 von Deutschland bzw. Österreich ohne Änderungen übernommen.

Obwohl die Normserie von der Wirtschaft begrüßt wurde und die Bedeutung durch das Einsetzen in bekannten Unternehmen stieg, wurde auch Kritik über

die Benutzerfreundlichkeit dieser Normen laut. Zum Beispiel hatten Softwareunternehmen und Anbieter von Dienstleistungen Probleme bei der Umsetzung der Normen. Der Grund dafür lag vermutlich daran, dass das zuständige technische Komitee größtenteils aus Experten des Produktionsbereiches bestand und somit Anforderungen von Software- und Dienstleistungsunternehmen unberücksichtigt blieben.

Aufgrund dieser Probleme wurden Leitfäden entwickelt, die bei der Anwendung der Normen behilflich sein sollten. Zum Beispiel ist ISO 9000-3 der Leitfaden für die Softwareentwicklung. Da in den letzten Jahren die Anzahl der Leitfäden stieg und die Anforderungen an die Normfamilie sich änderten, beschloss das TC/176 eine Weiterentwicklung der Normenserie.

Der Entwurf der Version 2000 entstand 1990 und nachdem er von den Mitgliedsländern der ISO akzeptiert wurde, wurde die DIN EN ISO 9000:2000 Normfamilie 1994 veröffentlicht. Diese Neufassung wurde im Dezember 2000 in Kraft gesetzt und seit Anfang 2004 ist nur noch eine Zertifizierung nach DIN EN ISO 9000:2000 möglich.

Die Zertifizierung wird von einer unabhängigen Zertifizierungsstelle vollzogen. Diese erfolgt durch ein Audit und ist drei Jahre, unter der Vorraussetzung, dass jährlich ein Überwachungs-Audit durchgeführt wird, gültig. Nach drei Jahren ist ein Wiederholungs-Audit erforderlich.

Aus Gründen der einfacheren Lesbarkeit wird „DIN EN ISO“ bzw. „ÖNORM EN ISO“ in dieser Arbeit mit „ISO“ abgekürzt und die Versionsangabe erfolgt nachgestellt, getrennt durch einen Doppelpunkt, z.B. ISO 9000:1994.

2.4 Aufbau der Normfamilie, Version 1987

Die ISO 9000 Normfamilie besteht aus Leitfäden und Nachweisstufen. ISO 9000 und 9004 sind Leitfäden zur Einführung und Dokumentation eines QM-Systems. Sie bestehen aus mehreren Teilen, welche die verschiedenen Aspekte und Branchen abdecken. Beispielsweise ist ISO 9000-3 der Leitfaden für die Anwendung von ISO 9001 auf die Entwicklung, Lieferung und Wartung von Software.

ISO 9001 bis 9003 sind verschiedene Nachweisstufen die für eine Zertifizierung benötigt werden. Alle Nachweisstufen fordern ein dokumentiertes und gelebtes QM-System. Die Dokumentation muss die Normanforderungen abdecken und besteht aus einem Handbuch und Verfahrens- und Arbeitsanweisungen. [vgl. Funk00, S.7 ff]

- ISO 9000: Diese Norm erklärt grundlegende Qualitätskonzepte und deren Begriffe. Sie ist der Leitfaden zur Auswahl und Anwendung der Nachweisstufen ISO 9001, 9002, 9003.
- ISO 9001: Diese Norm ist die Nachweisstufe für Entwicklung, Produktion, Montage und Wartung. Sie ist die umfassendste der drei Nachweisstufen.
- ISO 9002: Sie ist die Nachweisstufe für Produktion und Montage.
- ISO 9003: Diese Norm bezieht sich nur auf die Qualitätssicherung in der Endprüfung.
- ISO 9004: Sie ist ein übergeordneter Leitfaden und soll bei der Auswahl von Qualitätssicherungs-Elementen behilflich sein. Es gibt 20 solcher Elemente, einige davon wären: Prozesslenkung, Schulung, Designlenkung, usw..

Grundsätzlich muss man aber sagen, dass die Normfamilie bis auf eine Reihe definierter Mindestanforderungen einen weitgehenden Spielraum bei der Erstellung eines QM-Systems zulässt.

2.5 Nachteile der Version 1987

Innerhalb kurzer Zeit wurde die Normenreihe durch internationalen Handel und die Wirtschaft akzeptiert und gewann an Bedeutung. Jedoch tauchten auch schnell Kritikpunkte auf. Es wurde z.B. bemängelt, dass

- die Benutzerfreundlichkeit der Normen sich nur auf Hardware produzierende Industriebereiche erstreckte. Anbieter von Software und Dienstleistungen hingegen hatten Probleme bei der Benutzung und Umsetzung.
- die Dokumentenanforderungen für kleinere Unternehmen nur schwer zu erfüllen waren.
- sich die Struktur der Normenreihe zu sehr an technischen Großunternehmen orientierte und es somit an Allgemeingültigkeit mangelte.
- die Verwendung der Begriffe „Lieferant“, „Einkäufer“ und „Kunde“ sehr verwirrend war.

Grundsätzlich sollte man aber auch erwähnen, dass es international üblich ist, eine Norm nach einem Zeitraum von fünf Jahren einem Review zu unterziehen und insofern war die Zeit reif für eine Neuauflage. [Thal96, S.33]

Deshalb beschloss das zuständige TC die Normfamilie weiterzuentwickeln. An die neue Version wurden zwei Forderungen gestellt.

1. Vorhandene Unklarheiten beseitigen.
2. Die Normfamilie zu vereinfachen.
3. Neue Entwicklungen berücksichtigen.

Um diese Forderungen zu verwirklichen, wurde ein 2-Phasen Modell für die Revision entwickelt.

Phase I: Kurzzeitrevision
Phase II: Langzeitrevision

[vgl. Schö01, S.21]

Im den nächsten Abschnitten wird auf den Inhalt der zwei Revisionen genauer eingegangen.

2.6 Kurzzeitrevision ISO 9000:1994

In der Kurzzeitrevision blieb die ursprüngliche Struktur erhalten. Das Ziel war die Behebung der Unklarheiten, Neudefinition von Begriffen und die Festlegung der Dokumentenanforderungen.

Die Bedeutung des Begriffes „Produkt“ wurde dahingehend erweitert, dass er nicht nur Hardware, sondern auch Software, Dienstleistungen oder Kombinationen daraus einschließt. [vgl. ISO194, S. 6]

Außerdem wurden die Begriffe „Lieferant“, „Einkäufer“ und „Kunde“ klarer voneinander abgegrenzt, da diese nicht immer einheitlich verwendet wurden und dadurch sehr verwirrend waren. Weiters wurde der Kundenzufriedenheit mehr Aufmerksamkeit geschenkt. [vgl. ISO494, S.8]

2.7 Langzeitrevision ISO 9000:2000

Die Gründe für die notwendige Revision wurden im Abschnitt 2.6 dargelegt. Aufbauend auf die Änderungen der Kurzzeitrevision wurden in der Langzeitrevision große Veränderungen betreffend Struktur und Inhalt vollzogen. Außerdem versuchte man eine bessere Umsetzung des Total Quality Management Gedankens zu erreichen. Auf den Vergleich zwischen ISO 9000 und dem Total Quality Management, kurz TQM, wird im letzten Abschnitt genauer eingegangen.

Ziele der großen Revision:

- Anwendbarkeit für Unternehmen aller Branchen und Größen
- die Übersichtlichkeit in der Normenfamilie soll erhöht und die Zahl der Normen reduziert werden
- Vereinfachung und Vereinheitlichung der verwendeten Begriffe in der Norm
- Das starre Prinzip des Nachweises der 20 QM-Elemente soll durch ein flexibles System abgelöst werden. Die im Betrieb real ablaufenden Prozesse sollen die Struktur vorgeben.
- Orientierung am Nutzen aller interessierten Parteien, und Verstärkung der Kundenorientierung
- Verbesserung der Kompatibilität und Integrationsfähigkeit mit anderen Managementsystemen (z.B. Umweltmanagementsystemen)
- Unterstützung von Bewertungsverfahren zur Eigenbewertung

[Brau02, S.17 ff.]

3 ISO 9000:2000

3.1 Aufbau der Normfamilie, Version 2000

Um die Normfamilie zu vereinfachen und dessen Übersichtlichkeit zu erhöhen, wurde folgende neue Struktur festgelegt.

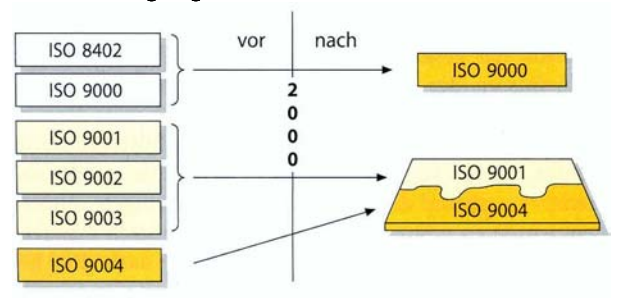


Abbildung 1: Struktur der Normfamilie [Brau02, Bild 2, S. 18]

ISO 8402 (Definition der Begriffe) und ISO 9000-1 (Leitfaden zur Auswahl und Anwendung) wurde durch ISO 9000 ersetzt.

ISO 9000-1 (Leitfaden für ISO 9001, 9002, 9003) wurde mit den Normen ISO 9001, 9002 und 9003 vereinigt und mit ISO 9001 benannt.

Der Leitfaden ISO 9004-1, der sich mit den Elementen eines QM-Systems befasste, wurde überarbeitet und durch die neue Norm ISO 9004 (Leitfaden zur Leistungsverbesserung) ersetzt.

Die Normen ISO 9000-3 (Leitfaden für Entwicklung, Lieferung und Wartung von Software), ISO 9004-2 (Leitfaden für Dienstleistungen) und die ISO 9004-3 (Leitfaden für verfahrenstechnische Produkte) sind entfallen.

Zusammenfassend ergibt sich folgender Aufbau:

ISO 9000:2000	Grundlagen und Begriffe
ISO 9001:2000	Anforderungen
ISO 9004:2000	Leitfaden zur Leistungsverbesserung
ISO 19011	Leitfaden für das Auditieren von Qualitätsmanagement- und Umweltsystemen

Durch diese Änderung wurden die 25 Einzelnormen mit einem Umfang von 1000 Seiten auf vier Hauptnormen mit einem Umfang von ca. 200 Seiten reduziert. [Schö01, Seite 31]

3.2 Prinzipien der neuen Version

Im Zuge der Revision wurden acht Prinzipien für das Qualitätsmanagement vorgestellt. Sie stellen die Basis der Forderungen und des Aufbaus der neuen ISO 9000 Version dar. Mit Hilfe dieser Prinzipien soll die

Leistungsfähigkeit eines Unternehmens verbessert werden.

1. Kundenorientierung. Ein Unternehmen ist hauptsächlich von seinen Kunden abhängig, deshalb sollte die gesamte Unternehmensstruktur auf die Kundenbedürfnisse ausgerichtet sein. Wünsche und Bedürfnisse des Kunden müssen verstanden und erfüllt werden. Es sollte danach gestrebt werden die Erwartungen des Kunden zu übertreffen.

2. Führung. Die Führung gibt die Richtung und den einheitlichen Zweck der Organisation vor. Aufgabe der Führung ist es auch ein internes Umfeld zu schaffen und zu erhalten, in dem die Mitarbeiter sich voll auf das Erreichen der Ziele der Organisation konzentrieren können. Wichtig ist die Formulierung einer klaren Unternehmensvision und das Leiten durch Vorbild.

3. Beteiligung der Mitarbeiter. Die Mitarbeiter sind das Herz einer Organisation. Um ihre Fähigkeiten zum Vorteil der Organisation zu nutzen, ist die Einbeziehung in das Organisationsgeschehen wichtig. Weiters sollen die Mitarbeiter ermutigt werden, aktiv nach Verbesserungsvorschlägen zu suchen.

4. Prozessorientierung. Ein erwünschtes Ergebnis wird effizienter erreicht, wenn die betroffenen Ressourcen und Aktivitäten als Prozesse geleitet werden. Es können Teilschritte, Schnittstellen, Ein- und Ausgaben identifiziert werden.

5. Systemorientierung. Es ist wichtig das Unternehmen als Gesamtsystem von zusammenhängenden Prozessen zu verstehen, da eine Organisation immer als ein Ganzes funktioniert. Das Verstehen und Managen dieses Systems von Prozessen erhöht die Effizienz einer Organisation beim Erreichen ihrer Ziele.

6. Ständige Verbesserung. Da Qualität keine statische sondern eine dynamische Größe ist, muss ein Interesse eines Unternehmens die ständige Verbesserung und Weiterentwicklung der Produkte und Prozesse sein.

7. Sachliche Entscheidungsfindung. Um wirksame Entscheidungen treffen zu können, müssen Daten und Informationen gesammelt und analysiert werden.

8. Lieferantenbeziehung zum gegenseitigen Vorteil. Eine Organisation und ihre Lieferanten sind voneinander abhängig. Die Verbesserung der Beziehungen zum beiderseitigen Vorteil steigert die Wertschöpfung beider Unternehmen.

4 ISO 9000 im Vergleich

4.1 TQM (Total Quality Management)

Unter TQM versteht man ein umfassendes Konzept in dem das zentrale Ziel die Qualität aus der Sicht des Kunden ist. Ein solches Konzept entsteht nicht von alleine, alle Beteiligten müssen aktiv mitarbeiten und es muss der TQM Ansatz „gelebt“ werden. Die wichtigsten Prinzipien sind die ständige Verbesserung, die Kundenorientierung, das Kunden-Lieferanten-Verhältnis und die Prozessorientierung. [Balz98, S. 339 ff.]

4.1.1 TQM vs. ISO 9000

Die ISO 9000:1994ff. beinhalteten nur Teile der TQM Prinzipien. In der neuen Version der ISO 9000 Normfamilie wurden einige wichtige Prinzipien von TQM berücksichtigt und somit ein Schritt in Richtung TQM gegangen. Unter diesen Aspekten sind vor allem die Kunden-, Mitarbeiter-, Prozessorientierung und die ständige Qualitätsverbesserung zu nennen. Die soziale Komponente wird jedoch auch weiterhin bei TQM stärker behandelt. Dem gegenüber ist die ISO 9000 besser fassbar und leichter in ein Unternehmen einzuführen, da nicht die Unternehmenskultur geändert werden muss. [Balz98, S. 353 ff.]

4.2 CMM (Capability Maturity Model)

In diesem System werden fünf Qualitätsstufen des Software-Entwicklungsprozesses unterschieden. Diese Stufen bauen aufeinander auf und beschreiben einen bestimmten Reifegrad des Entwicklungsprozesses. Die Reifegrade reichen von der erste Stufe, die einem chaotischen, nicht reproduzierbaren Entwicklungsprozess entspricht, bis zur Stufe fünf, die auf einen optimierten, sich ständig verbessernden Entwicklungsprozess hinweist. [Balz98, S. 262 ff.]

4.2.1 CMM vs. ISO 9000

- Der CMM-Ansatz konzentriert sich hauptsächlich auf die Qualitäts- und Produktivitätssteigerung des Software-Entwicklungsprozesses, hingegen kann ISO 9000 für verschiedene Branchen verwendet werden.
- ISO 9000 ist ein fester Industriestandard, CMM ist ein Hilfsmittel zur Problemanalyse und Prozessverbesserung ist.
- Beide Ansätze enthalten unabhängige Audits, umfangreiche Dokumentation und schließen mit einem Zertifikat ab.
- Es existieren kaum Erfahrungswerte von hoch eingestuften Unternehmen in CMM, im Gegensatz zur weit verbreiteten ISO 9000

Daraus ergibt sich, dass man sich nicht zwischen ISO 9000 und CMM entscheiden muss, sondern dass CMM im Bereich der Softwareentwicklung eine Ergänzung darstellt. Für kleine und mittlere Software-Unternehmen wird jedoch die Einführung von ISO 9000, aufgrund des Umfanges und den Anforderungen, einfacher sein. [Balz98, S. 373 ff.]

5 Zusammenfassung

In dieser Arbeit wurden grundsätzlich die Vorteile eines Qualitätsmanagementsystems dargelegt, da Qualitätsorientierung ein Wettbewerbsvorteil für moderne Unternehmen ist. Wir sind der Meinung, dass Aufwand und Kosten für eine Zertifizierung nicht gescheut werden sollen, da man dadurch das Vertrauen der Kunden und Lieferanten steigern und das Image des Unternehmens verbessern kann. Eine Zertifizierung alleine führt nicht zum Erfolg, sondern Qualität muss im Unternehmen von allen Beteiligten „gelebt“ werden.

Die ISO 9000 besitzt diese grundlegenden Vorteile eines Qualitätsmanagementsystems und zählt darüber hinaus zu den bekanntesten Standards, da es in der Wirtschaft häufig verwendet wird. In der neuen Version der ISO 9000 wurden viele Nachteile behoben, sodass eine leichtere Anwendung für alle Unternehmensgrößen und Branchen möglich ist.

Ist das Ziel eines Unternehmens TQM, dann kann man sich mit einer ISO 9000 Zertifizierung diesem Ziel nähern, da in der Langzeitrevision einige wichtige Prinzipien des TQM-Konzepts umgesetzt wurden. Auch in der Softwarebranche gelten alle Vorteile der ISO 9000 Zertifizierung. Der Prozessverbesserung wurde in der Weiterentwicklung der Norm mehr Aufmerksamkeit geschenkt, trotzdem kann sie CMM als speziell entwickeltes Modell für die Verbesserung des Softwareentwicklungsprozesses nicht ersetzen.

Die ISO 9000 Normfamilie ist eine gute Wahl für Qualitätsmanagement und sie schließt andere Modelle nicht aus, sondern unterstützt sie.

6 Literaturverzeichnis

[Thal01] THALLER Georg Erwin, ISO 9001:2000 Software-Entwicklung in der Praxis, Verlag Heinz Heise, Hannover, 2001

[Funk00] FUNKE Thomas, Softwareentwicklung in mittelständische Unternehmen mit ISO 9000, Springer Verlag, Berlin, 2000

[Brau02] BRAUER Jörg-Peter, DIN EN ISO 9000:2000ff. umsetzen, Carl Hanser Verlag, München Wien, 2002

[Balz98] BALZERT Helmut, Lehrbuch der Software-Technik, Spektrum Akademischer Verlag GmbH, Heidelberg – Berlin, 1998

[Thal96] THALLER Georg Erwin, ISO 9001: Software-Entwicklung in der Praxis, Verlag Heinz Heise, Hannover, 1996

[Glaa93] GLAAP Winfried, ISO 9000 leicht gemacht, Carl Hanser Verlag, München Wien, 1993

[Schö01] SCHÖBEL Sven, Die Revision der DIN ISO 9000:1994 zur DIN ISO 9000:2000 im Vermessungswesen, Diplomarbeit, Hochschule für Technik und Wirtschaft Dresden (FH), Dresden, 2001

[Schn97] SCHNEIDER Dieter, Modernes Qualitätsmanagement durch ISO 9000?, Diplomarbeit, Universität Klagenfurt, 1997

[ISO094] ÖNORM EN ISO 9000-1, Qualitätsmanagementsysteme - Leitfaden zur Auswahl und Anwendung, ON – Österreichisches Institut für Normung, Wien, 1994

[ISO194] ÖNORM EN ISO 9001, Qualitätsmanagementsystem - Anforderungen, ON – Österreichisches Institut für Normung, Wien, 1994

[ISO494] ÖNORM EN ISO 9004-1, Qualitätsmanagement und Elemente eines Qualitätssicherungssystems, ON – Österreichisches Institut für Normung, Wien, 1994

Requirements Engineering in globalen Projekten

Marion Kury
9960541
mkury@edu.uni-klu.ac.at

Abstract

Die allgemeine Globalisierung macht auch vor Softwareprojekten keinen Halt. Immer mehr Projekte werden von Projektteams bearbeitet, die über die ganze Welt verteilt sind. Diese neue Gegebenheit hat Auswirkungen auf die gesamte Projektentwicklung, betrifft aber besonders die Requirements Engineering Phase, da dies die kommunikationsintensivste Phase des gesamten Projektes ist. Diese Arbeit befasst sich mit den Herausforderungen, mit denen sich die Requirements Engineering Phase in globalen Projekten konfrontiert sieht. Die geographische Distanz führt zu verschiedenen Problemen, die sonst nicht auftauchen und verstärkt aber andererseits auch Probleme die während der Requirements Engineering Phase in jedem Projekt auftreten. In dieser Arbeit werden die auftretenden Probleme beschrieben und Lösungsansätze vorgestellt. Da jedoch für etliche Probleme noch kein Lösungsansatz vorhanden ist, wird weitere Forschung in dieser Richtung angeregt. Eine, auf die neuen Gegebenheiten abgestimmte, Requirements Engineering Methode ist zu erarbeiten.

1. Motivation

Die allgemeine Globalisierung schreitet voran. Viele Unternehmen haben ihre Abteilungen über die ganze Welt verteilt. Davon ist auch die Softwareindustrie nicht ausgenommen. Softwarehersteller führen Projekte durch, bei denen verschiedene Projektgruppen an unterschiedlichen Standorten arbeiten. Dies stellt eine Herausforderung für die gesamte Projektentwicklung dar. Eine besonders betroffene Phase der Projektentwicklung stellt dabei die Requirements Engineering Phase dar. Dies ist die Phase der Anforderungsermittlung. Auf den Anforderungen basiert das gesamte Projekt. Somit ist der Erfolg des gesamten Projektes von den Ergebnissen der Requirements Engineering Phase abhängig. Deshalb ist es wichtig, sich damit auseinander zu setzen, welche Probleme die geographische Distanz der verschiedenen Projektteams hervorruft. Diese Arbeit wird etliche Probleme auflisten die dabei zu lösen sind und – soweit vorhanden – Lösungsmöglichkeiten aufzeigen.

2. Probleme

Dieser Abschnitt befasst sich mit der Definition der Probleme die in globalen Projekten, in der Requirements Engineering Phase auftreten.

2.1. Kommunikationsbarrieren

Durch geographische Distanz entstehen Schwierigkeiten während der Requirements Engineering Phase. Diese Phase erfordert, verglichen mit anderen Phasen, überdurchschnittlich viel Kommunikation. Üblicherweise werden in vielen persönlichen Treffen der verschiedenen Stakeholder die fachlichen und technischen Anforderungen des Projektes ermittelt. Bei globalen Projekten muss für jede Kommunikation auf technische Hilfsmittel zurückgegriffen werden. Bei Kommunikation mit Hilfe von technischen Hilfsmitteln, geht ein Teil der nonverbalen Information verloren. Diese Information ist aber gerade beim Ausarbeiten von Kompromissen wichtig, was einen nicht unwesentlichen Teil der Requirements Engineering Phase ausmacht.

Eine weitere Erschwernis ist die Planung der Kommunikation. Spontane oder zufällige Treffen sind bei globalen Projekten unmöglich. Wenn ein Problem auftaucht, muss erst ein Termin gesucht werden, was durch die Zeitverschiebung (siehe Abschnitt 2.3) noch zusätzlich verkompliziert wird. Die Möglichkeit, den Partner sofort aufzusuchen, um das Problem zu lösen besteht nicht. Bei Projekten mit großer Zeitverschiebung, wenn keine gemeinsamen Bürostunden vorhanden sind, ist sogar mit einem Werktag Verzögerung zu rechnen. Somit verlängert sich die Requirements Engineering Phase, die oft auch

bei lokalen Projekten als zu zeitaufwendig empfunden wird.

Eine zusätzliche Kommunikationsbarriere ist die Fremdheit der einzelnen Personen untereinander. Wenn Unsicherheiten auftauchen, ist die Hemmschwelle bei unbekannten Personen nachzufragen höher. Außerdem besteht auch die Gefahr, dass nicht bekannt ist welche Person aus dem anderen Team mit dem Gebiet betraut ist. Damian gibt in ihrer Arbeit [5] zu bedenken, dass die Entfernung auch gewisse Ressentiments stärken kann. Wenn keine Teambildung über die Grenzen hinweg erfolgt, besteht große Gefahr, dass jedes Team versucht für sich selbst die beste Lösung zu finden, statt die beste Lösung für das Projekt zu suchen.

Diese Herausforderungen bedingen, dass die Arbeit eines Requirements Engineers in globalen Projekten anspruchsvoller ist, als bei lokalen Projekten. Der Requirements Engineer muss ein größeres Maß an Koordinationsfähigkeit mitbringen und auch die Fähigkeit die wichtigen Fragen im Fokus zu behalten. Dinge, die sonst vielleicht über informelle Kommunikation kolportiert werden, müssen explizit angesprochen und geklärt werden.

2.2. Wissensmanagement

Wie im vorigen Abschnitt besprochen ist Kommunikation ein wichtiges Thema während der Requirements Engineering Phase. Das liegt daran, dass das Wissen der einzelnen Stakeholder gesammelt und koordiniert werden muss. Große Teile des benötigten Wissens sind nirgendwo dokumentiert, sondern befinden sich laut Zowghi [2] nur in den Köpfen der jeweiligen Personen. Erst die gesammelte Menge an Information macht es schlussendlich möglich eine Definition der benötigten Anforderungen zu erstellen, die den Projektzielen tatsächlich entspricht. Dieses undokumentierte Wissen wird teilweise auf informelle Weise zwischen den verschiedenen Teilnehmern ausgetauscht. In einem globalen Projekt müssen die Informationen formaler verwaltet werden. Es ist dafür zu sorgen, dass das Wissen einzelner dokumentiert und allen anderen zu Verfügung gestellt wird.

2.3. Zeitverschiebung

Ein weiterer Einflussfaktor ist die Zeitverschiebung. Synchrone Kommunikation kann nur stattfinden, wenn beide Teams anwesend sind. Bei einer größeren Zeitverschiebung sind diese Zeiten jedoch spärlich bis überhaupt nicht vorhanden. Im Extremfall gibt es keine gemeinsamen Bürostunden. Somit ist synchrone Kommunikation auf vorher festgelegte Zeiten beschränkt. Ein Team, das an dieser Kommunikation teilnimmt, muss dafür Überstunden machen. Dies kann zu Ressentiments gegenüber dem anderen Team führen, die erst abgebaut werden müssen.

Asynchrone Kommunikation ist jederzeit möglich, erfordert jedoch Zeit. Eine Antwort ist erst am nächsten Werktag zu erwarten. Ein weiterer Nachteil von asynchroner Kommunikation, ist die damit verbundene Qualitätsminderung. Verständnisprobleme erfordern Zwischenfragen, die bei asynchroner Kommunikation unmöglich sind. Das Ausarbeiten von Kompromissen ist mit asynchroner Kommunikation zeitlich unmöglich, da es eine allmähliche Annäherung der Standpunkte erfordert.

2.4. Kulturelle Probleme

In diese Kategorie fallen nicht nur Unterschiede in der gelebten Kultur und Sprachprobleme, sondern auch Unterschiede in der Unternehmenskultur der verschiedenen Projektteams.

Einer der heikelsten Punkte bei verteilten Projekten ist die Sprachbarriere. Die Projektteilnehmer müssen in der Lage sein fachliche Diskussionen in einer für sie fremden Sprache zu führen. Dies erfordert nicht nur gute allgemeine Kenntnisse in der Sprache, sondern auch das entsprechende Fachvokabular. Die Sprachbarriere kann gerade bei Fachausdrücken zum Problem werden, siehe den Erfahrungsbericht von Johnston, Peters Schneider und Wellen [4]. In diesem Bericht stellt die Sprachbarriere ein Hindernis dar. Ein Konflikt zwischen zwei Teams, aus verschiedenen Ländern, ließ sich erst durch persönlichen Kontakt beilegen, als die Teammitglieder feststellten, dass die verschiedenen Termini die sie verwendeten, das gleiche bedeuten.

Ein weiterer potentieller Konfliktpunkt ist der unterschiedliche kulturelle Hintergrund der verschiedenen Projektteams. Das Auftreten von Kulturunterschieden wird oft unterschätzt. Teams müssen nicht von verschiedenen Kontinenten stammen, um unterschiedliche Kulturen zu repräsentieren. Bei einem österreichisch-schweizerischen Projekt kam es zu inkompatiblen Hilfetexten, da es in der Schweiz kein „ß“ gibt, siehe Led [3]. Natürlich steigen die kulturellen Unterschiede mit der Entfernung der Länder, aus denen die Projektmitarbeiter kommen.

Die im jeweiligen Land üblichen Umgangsformen und Hierarchien müssen gewahrt werden. Kulturelle Beschränkungen müssen allgemein bekannt gemacht werden. So ist in streng islamischen Ländern die Gebetszeit zu respektieren. Falls es zu Konflikten zwischen den Umgangsformen der einzelnen Kulturen

kommt, muss ein Kompromiss erarbeitet und vor allem auch publik gemacht werden. Es ist wichtig, dass sich alle beteiligten Kulturen akzeptiert und angenommen fühlen. Wenn nicht, führt das zu Unzufriedenheit, die sich schlussendlich im Ergebnis der Arbeit wieder spiegelt.

3. Lösungsansätze

Dieser Abschnitt wird sich damit befassen Lösungsmöglichkeiten für die im vorigen Abschnitt beschriebenen Probleme vorzustellen.

3.1. Knowledge Scouts

Als Lösungsmöglichkeit der Probleme Fremdheit und kulturelle Unterschiede erweisen sich die von Dutoit, Johnstone und Bruegge [1] als "Knowledge Scouts" bezeichneten Mitarbeiter. Diese Mitarbeiter arbeiten eine kurze Zeitspanne in einem Projektteam an einem anderen Standort. Dabei lernen sie die dortigen Mitarbeiter sowie die dortige Kultur kennen. Dadurch werden Kommunikationsbarrieren überwunden und ein globales Teamgefühl erzeugt. Diese Knowledge Scouts bilden die Bindeglieder der einzelnen Teams untereinander. Sie bringen dem fremden Team die Arbeitsweise und die Mitglieder ihres eigenen Teams näher. Nach ihrer Rückkehr geben sie ihr Wissen über das fremde Projektteam ihren Kollegen weiter. Dieses gegenseitige Kennen lernen fördert die Kommunikation zwischen den Teams. Nach dem Besuch des Knowledge Scouts steigt die Kommunikation zwischen den beiden Teams und somit auch die Qualität der Arbeit.

3.2. Standards

Standards werden in allen Projekten definiert. Ihre besondere Bedeutung in globalen Projekten bekommen sie, da weder die formelle noch die informelle Kommunikation zwischen den einzelnen Projektteams Zeit hat, sich allmählich zu entwickeln. Es ist ratsam auch Standards für verschiedene Kommunikationsarten zu definieren. Ein Standardlayout für E-Mails senkt die Hemmschwelle eine elektronische Kommunikation zu initiieren und hilft somit die Lösung des jeweiligen Problems schneller und exakter zu ermitteln. Jeder Projektteilnehmer muss wissen auf welche Art und Weise er, mit einem Kollegen vom anderen Projektteam, in Kontakt treten soll.

Natürlich müssen auch die zu erstellenden Dokumente ein Standardformat haben. Dadurch kann der Abgleich und die Integration der an verschiedenen Orten erstellten Dokumente problemloser erfolgen. Die Erstellung von UML-Use-Case Diagrammen von einem Team lässt sich nur schwer mit den natürlich-sprachlichen Dokumenten des anderen Teams abgleichen, siehe Johnston, Peters, Schneider und Wellen [4]. Diese Schwierigkeiten können durch die rechtzeitige Definition von Standards vermieden werden.

3.3. Tools

Für verteilte Projekte bietet sich die Verwendung von Tools besonders an, da dies eine Möglichkeit ist eine Einheit zu schaffen, die alle Teams gleichermaßen betrifft. Ein Requirements Engineering Tool zu verwenden ist auch eine einfache Methode gewisse Standards festzulegen. Die Dokumente haben dadurch ein vorgegebenes einheitliches Layout. Leider sind bisher noch keine Requirements Engineering Tools auf dem Markt, die die besonderen Anforderungen eines globalen Projektes berücksichtigen (Zowghi [2]).

Eine Erleichterung des Dokumentenmanagement und auch der projektinternen Kommunikation bringt ein Groupware-Tool. Bereits dokumentiertes Wissen wird durch das Abspeichern in allgemein zugänglichen Dokumenten veröffentlicht. Das ermöglicht interessierten Mitarbeitern auch in Themengebiete Einblick zu bekommen, die sie nicht direkt betreffen. Dies bildet ein Äquivalent für informelle Kommunikation über Teamgrenzen hinweg. Weiters bieten diese Tools die Verwendung von Foren an, die als Diskussionsplattformen genutzt werden können. Dieses Tool verbindet alle Mitarbeiter, egal ob sie in gleichen oder verschiedenen Projektteams arbeiten und bietet allen die selben Informationsmöglichkeiten

4. Konclusio

Die Requirements Engineering Phase hat in globalen Projekten mit zusätzlichen Herausforderungen zu kämpfen. Einige Lösungsansätze wurden im vorigen Abschnitt vorgestellt. Vorschläge, wie an die Lösung des Wissensmanagementproblems herangegangen werden soll, sind noch nicht dokumentiert. Für manches, wie die Zeitverschiebung und die kulturellen Unterschiede, gibt es keine Lösung, sondern nur mehr oder weniger effiziente Methoden mit den Gegebenheiten umzugehen.

Der Requirements Engineering Prozess eines globalen Projektes erfordert ein besonderes Fingerspitzengefühl. Viele Schwierigkeiten die in jedem Projekt auftreten, werden verschärft, wie z.b.: Kommunikationsprobleme und Wissensmanagement.

Andere Probleme treten zusätzlich auf, wie die Zeitverschiebung und die kulturellen Unterschiede.

Der Requirements Engineering Prozess eines globalen Projektes ist ein Gebiet auf dem noch viel zu forschen ist. Es müssen Requirements Engineering Methoden entwickelt werden, die auf all diese neuen Herausforderungen eingehen.

5. Zusammenfassung

Die Requirements Engineering Phase in globalen Projekten ist ein äußerst spannendes und aktuelles Thema. In dieser Phase treten viele Probleme, die es bei globalen Projekten gibt, besonders hervor, andere kommen hinzu. Die Probleme wurden in dieser Arbeit aufgelistet. Die Schwierigkeiten liegen zum großen Teil in der Erschwernis der Kommunikation. Deshalb nimmt das Kapitel Kommunikationsbarrieren einen so großen Platz in dieser Arbeit ein. Es wird auf die Probleme der Sprachkenntnisse, der Fremdheit, synchroner und asynchroner Kommunikation eingegangen. Des weiteren werden die Probleme Wissensmanagement, Zeitverschiebung und der Kulturunterschiede angesprochen. In diesem Dokument wurden auch einige Lösungsmöglichkeiten aufgezeigt. Die von Dutoit, Johnstone und Bruegge erwähnten Knowledge Scouts sind ein wichtiger Schritt, da sie die soziale Komponente berücksichtigen, die in der Requirements Engineering Phase eine wichtige Rolle spielt.

Eine weitere Hilfe ist die Definition von Standards, da in globalen Projekten ein größerer Teil des Informationsaustausches über Dokumente erfolgen muss, als bei lokalen Projekten. Ein einheitliches Layout beschleunigt die intellektuelle Verarbeitung, und erleichtert das Auffinden von Differenzen in verschiedenen Dokumenten.

Die Verwendung von Groupware-Tools ist für verteilte Projekte unabdingbar, da sie für eine Vernetzung sorgt und teilweise auch Kommunikationsstrukturen vorgibt. Sie bieten die technische Kommunikationsmöglichkeit, die in globalen Projekten gebraucht werden, sowie einen gemeinsamen Platz für den Dokumentenaustausch.

Diese Lösungsansätze können jedoch nur der Anfang sein. Viele Probleme sind noch offen. Die Requirements Engineering Phase in globalen Projekten bietet noch ein weites Forschungsfeld, auf dem hoffentlich bald weitere Erkenntnisse gefunden werden.

6. References

[1] Allen H. Dutoit, Joyce Johnstone, Bernd Bruegge, “Knowledge scouts: Reducing communication barriers in a distributed software development project”, 8th Asia-Pacific Software Engineering Conference (APSEC 2001), Macau China, Dec. 2001

[2] Zowghi Didar, “Does Global Software Development Need a Different Requirements Engineering Process?”, International Workshop on Global Software DevelopmentOrlando, Florida, May 21, 2002

[3] Led, Vera, “Kritische Erfolgsfaktoren in den ‘Front-End-Lifecycle-Phasen’”, Diplomarbeit, 1999

[4] Johnston L., Peters D., Schneider J., Wellen U., “Requirements Analysis in Distributed Software Engineering Education: An Experience Report“, 6th Australian Workshop on Requirements Engineering (AWRE,2001),Sydney, November 2001

[5] Damian Daniela, “The study of requirements engineering in global software development: as challenging as important”, International Workshop on Global Software DevelopmentOrlando, Florida, May 21, 2002

Die "Goal Question Metric" Methode

Jakab Michael
9960117
mjakab@aon.at

Ofner Michael
9960295
crashproof@gmx.at

Abstract

Mit Hilfe der GQM (Goal Question Metric) Methode wird danach getrachtet, eine prozessübergreifende, kontextspezifische Auswahl von geeigneten Metriken zu erreichen, die zur Evaluierung und zur Erlangung von Feedback aus speziellen Prozessen herangezogen werden kann[3]. Die dafür nötigen Kenntnisse werden in dieser Arbeit theoretisch und auch anhand eines praktischen Beispiels illustriert. Im praktischen Bezug ist auch die Betrachtung der in den Phasen der GQM Methode anfallenden Kosten, die in dieser Arbeit ebenfalls behandelt werden, von großem Interesse.

Um dieses Verfahren in gegenwärtig existierende, qualitätssichernde Standards einordenbar zu machen, wird der qualitätssteigernde Effekt von GQM, bei Anwendung auf bestimmte CMM Ebenen – wie er von Pfleeger in [5] gesehen wird – beschrieben.

1. Einleitung

Die Software Entwicklung benötigt zur Gewinnung von Feedback bzw. Evaluierung der Qualität ihrer Arbeit, der Kosten, der Risiken und des Fortschrittes einen geeigneten Mechanismus [1]. Dieser, die Softwaremessung, ist der Prozess, durch welchen Werte oder Symbole, Attributen oder Entitäten in der realen Welt, zugeordnet werden, sodass sie klar definierten Regeln entsprechen [3]. Dabei resultieren die Werte oder Symbole aus der Anwendung von „Software Metriken", wobei eine Messung mehrere Metriken und eine Metrik mehrere Werte umfassen kann.

In der Literatur ist eine große Anzahl von Metriken und Kategorisierungen für diese angeführt. Keine dieser Metriken für sich richtet sich aber direkt auf die drei Hauptziele, die durch Messung gestützt werden sollen. Diese wären [3]:

- *Verstehen* – Die verschiedenen Aspekte von Prozessen und Produkten verstehen und dokumentieren, um ein besseres Verständnis des Verhältnisses von Aktivitäten zu Entitäten zu bekommen.
- *Kontrollieren* – Die Erreichung der festgelegten Ziele validieren, um bei Abweichungen korrigierend eingreifen zu können.
- *Verbessern* – Durch die Analyse der Daten aus den Bereichen „Verstehen" bzw. „Kontrollieren" Verbesserungsmöglichkeiten identifizieren und anwenden.

Es zeigt sich, dass es notwendig ist, eine kontextspezifische Auswahl an Metriken zu treffen, um die oben angeführten Ziele abdecken zu können.

Daneben zeigen viele Studien, die sich mit der Anwendung von Metriken und Modellen im industriellen Umfeld befassen, dass effektive Messungen

- auf bestimmte Ziele fokussiert sein müssen,
- auf alle Produkte, Prozesse und Ressourcen angewendet werden müssen, als auch
- mit den unternehmensspezifischen Charakteristika und Zielen interpretiert werden müssen.

Das heißt, dass die Messung in „top-down" Manier zu definieren ist und ferner eine Fokussierung auf die Ziele und Modelle gegeben sein muss. [1]

GQM (Goal Question Metric) ist eine Vorgehensweise, die diesem Ziel getriebenem „Top-Down" Modell folgt und nach diesen Kriterien zu den im gewählten Kontext relevanten Metriken führt.

2. „Goal Question Metric" Methode

Der GQM Ansatz basiert auf der Annahme, dass eine Organisation, welche zielgerichtete Messungen durchführen möchte, zuerst die Ziele für sich selbst und ihre Projekte festlegen muss. Ausgehend von diesen müssen in weiterer Folge die Daten, welche diese Ziele definieren sollen, identifiziert werden und basierend auf diesen ein Rahmenwerk zur Interpretierung geschaffen werden. Es ist also wichtig, die Informationsanforderungen, die eine Organisation hat, zu identifizieren, sodass diese quantifiziert werden können, um zu analysieren, ob die Ziele erreicht wurden. [1]

2.1. Ebenen der GQM Methode

Das GQM Modell inkludiert drei verschiedene E-benen, siehe Abbildung 1. Es beginnt nach einem „Top-Down“ Schema, indem zuerst ein spezielles Ziel (Goal) definiert wird, aus welchem dann mehrere Fragen (Question) abgeleitet werden können, die dieses Ziel definieren sollen. Die Metriken (Metric) werden so ausgewählt, dass die daraus gewonnenen Daten die im vorhergehenden Schritt definierten Fragen beantworten können. Aus der Auswertung der Fragen/Antworten kann im letzten Schritt festgestellt werden ob bzw. inwiefern die Ziele erreicht wurden.
Die folgenden Informationen stammen aus [1].

Abbildung 1 - GQM Modell

Konzeptuelle Ebene (Goal)

Aus einer Vielzahl von Gründen wird ein Ziel für ein bestimmtes Objekt definiert. Dieses kann auf unterschiedliche Qualitätsmodelle Bezug nehmen und verschiedene Bedürfnisse und Sichtweisen berücksichtigen.

Objekte, auf welche sich die Ziele beziehen können, sind:

- Produkte: Artefakte, Arbeitsergebnisse und Dokumente die während des „System life cycle“ erstellt werden. z.B. Spezifikationen, Designs, Programme, Testumgebungen
- Prozesse: Software bezogenen Aktivitäten. z.B. Spezifizieren, Designen, Testen, Interviewen
- Ressourcen: Mittel, welche von Prozessen gebraucht werden, um einen Output zu generieren. z.B. Personal, Hardware, Software, Büroräumlichkeiten

Die gewonnenen Ziele dienen als Ausgangspunkt für den GQM Ansatz.

Operative Ebene (Question)

Eine Menge von Fragen charakterisiert die Art, wie ein gewisses Ziel evaluiert werden kann. Sie dienen also zur Feststellung, ob ein gewisses Ziel erreicht wurde. Die Fragen versuchen das Objekt (Produkt, Prozess, Ressource) hinsichtlich der selektierten Qualitätsanforderungen zu definieren. Dabei ist zu beachten, dass die Definition den gewählten Gesichtspunkt bzw. die gewählte Thematik berücksichtigt.

Quantitative Ebene (Metric)

Konkret gemessene Werte werden den Fragen der jeweiligen Ziele zugeordnet, um eine quantitative Antwort zu liefern. Eine Metrik kann hier für mehrere Fragen relevant sein. Die gemessenen Werte sind:

- Objektiv: Wenn sie nur von dem quantitativ zu erfassenden Objekt abhängen und nicht von der Sichtweise aus der sie betrachtet werden. z.B. Anzahl der Versionen eines Dokuments, Arbeitsstunden für eine gewisse Aufgabe, Größe des Programms.
- Subjektiv: Wenn sie sowohl von dem Objekt als auch der Sichtweise abhängen. z.B. Lesbarkeit von Text, Grad der Kundenzufriedenheit.

2.2. Phasen der GQM Methode

Die GQM Methode wird in 4 Phasen aufgeteilt, siehe Abbildung 2.

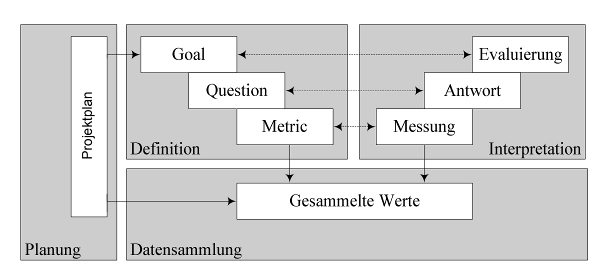


Abbildung 2 - 4 Phasen der GQM Methode [6]

Planungsphase

In dieser Phase sollen die Voraussetzungen für die erfolgreiche Durchführung eines GQM Messprogramms geschaffen werden. Hierzu zählt, neben der Motivation und Ausbildung der Mitarbeiter, die Erstellung eines Projektplans. Dieser Projektplan dokumentiert die Prozeduren, die Zeiteinteilung und das Gesamtziel des geplanten Projektes.

Definitionsphase

Diese Phase beinhaltet all jene Aktivitäten, die zur formellen Definierung eines Messprogramms notwendig sind. Als Ergebnis werden mehrere Dokumente erstellt [6]:

- der GQM Plan
- der Messplan
- der Analyseplan

Das Definieren von Zielen ist kritisch für die erfolgreiche Anwendung des GQM Ansatzes und wird von spezifischen methodischen Schritten unterstützt. Wie aus Abbildung 3 ersichtlich, besteht ein Ziel (GOAL) aus mehreren Koordinaten.

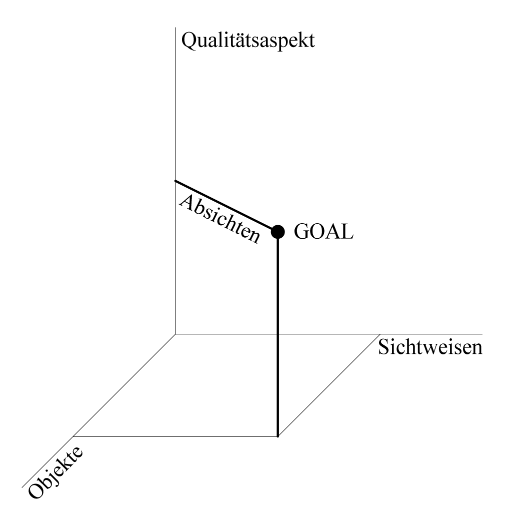


Abbildung 3 – Koordinaten eines Goals [1]

Es zeigt sich also, dass ein „Goal“ aus drei Koordinaten bestimmt wird (Qualitätsaspekt, Objekt, Sichtweise).

Die erste dieser Koordinaten leitet sich aus der Unternehmenspolitik her. Durch die Politik und Strategie eines Unternehmens, aber auch durch das Interviewen von relevanten Mitarbeitern, lassen sich der Qualitätsaspekt und die Absicht eines „Goals“ definieren.

Als zweite Informationsquelle dienen die Beschreibungen und das Wissen über unsere Prozesse und Produkte. Hieraus wird das Objekt unseres „Goals“ so exakt wie möglich formal definiert.

Die Sichtweise lässt sich durch unser Wissen über die Organisation bzw. Struktur der Organisation hinsichtlich unterschiedlicher Interessen ableiten.

Durch Kombination der drei oben genannten Informationsquellen kann nun ein sinnvolles und exakt definiertes „Goal“ bestimmt werden.

Die „Questions“, die nun ausgehend von den „Goals“ abgeleitet werden, stellen eine Verfeinerung dieser dar. Während die „Goals“ sehr abstrakt definiert sind, sollen die „Questions“ diese konkretisieren und die Möglichkeit zur Interpretation geben. Durch Beantwortung der „Questions“ soll schlussendlich festgestellt werden, ob ein definiertes „Goal“ erreicht wurde. Es ist also wichtig, während der Erstellung der „Questions“, immer wieder zu verifizieren, ob sie wirklich zur Evaluierung des „Goals“ beitragen.

Auf Basis der abgeleiteten „Questions“ werden adäquate Metriken bestimmt, welche die quantitativen Informationen liefern, durch die die „Questions“ beantwortet werden können. Bei der Auswahl der Metriken sollte darauf geachtet werden, wie schwierig es ist, diese in den Projektablauf einzubinden.

Im GQM Plan ist der oben beschriebene Ablauf dokumentiert. Außerdem erstellt man in der Definitionsphase einen Messplan. Dieser definiert für jede der identifizierten Metriken die Art der Messung, die Personen, welche die Messung durchführen, die Tools, die zur Unterstützung verwendet werden und alle möglichen Werte, die durch eine spezifische Messung zurückgeliefert werden können.

Der Analyseplan simuliert schlussendlich die Interpretation des GQM Plans, bevor die tatsächliche Datensammlung durchgeführt wird. Dies ermöglicht es, Schwachstellen in der Definition des GQM Plans festzustellen und zu korrigieren.

Datensammlungsphase

In dieser Phase werden die Daten für die im GQM Plan definierten Metriken gesammelt und gespeichert. Wie diese Datensammlung vollzogen wird, wurde bereits in der vorhergehenden Phase durch den Messplan festgelegt. Dabei werden folgende Aspekte abgedeckt:

- Wer sammelt die Daten für eine bestimmte Metrik?
- Wann werden die Daten erhoben?
- Wie können die Daten am effektivsten und effizientesten gesammelt werden (Tools)?
- Wohin werden Daten zugeordnet?

Interpretationsphase

Nach Abschluss der Datensammlung werden die gesammelten Werte über den GQM Plan den zugehörigen „Questions“ zugeordnet. Hieraus kann evaluiert werden, ob oder inwiefern die definierten „Goals“ erreicht wurden. Abschließend sollte eine Kosten/Nutzen Analyse über das GQM Programm durchgeführt werden, um festzustellen, ob die Anwendung für die Organisation von Nutzen war.

2.3. GQM und CMM

Während einige Organisationen mit klar definierten Prozessen arbeiten, sind andere variabler und ändern diese signifikant mit den Personen, die gerade an einem Projekt beteiligt sind. Das CMM Modell bildet diesen Sachverhalt auf fünf Stufen (maturity levels) ab. Auf der niedrigsten Stufe (initial) sind kaum Informationen darüber vorhanden, wie ein spezieller Prozess abläuft. Je höher man in den Stufen des Modells aufsteigt, desto besser sind die Prozesse erfasst und definiert.

Wie aus Tabelle 1 ersichtlich, werden mit steigender Stufe und exakterer Definition der Prozesse mehr und mehr Sachverhalte erkennbar, wobei genau jene erkannten Gegebenheiten für einen Entwickler messbar sind. Nicht Sichtbares kann auch nicht gemessen werden.

Tabelle 1 - Prozess maturity und Metriken [5]

Maturity level	Charakteristiken	Typen von Metriken
Optimizing	Verbesserungsfeedback für den Prozess	Prozess mit Feedback zur Änderung
Managed	Gemessener Prozess	Prozess mit Feedback zur Kontrolle
Defined	Prozess definiert und institutionalisiert	Produkt
Repeatable	Prozess von Individuen abhängig	Projektmanagement
Initial	Ad hoc	Grundmetriken

Wird das GQM Modell zur Planung eines Messprogramms genutzt, macht es also Sinn, vor der Festlegung auf bestimmte Metriken, die „Maturity Stufe" der Organisation zu identifizieren. Das CMM Modell suggeriert, wie oben bereits ausgeführt, dass nur sichtbare Sachverhalte gemessen werden können. Die Verwendung von CMM in der GQM Definitionsphase zeigt also ein umfassenderes Bild davon, welche Metriken am meisten Nutzen bringen. GQM hilft uns also zu verstehen, warum wir ein bestimmtes Attribut messen und CMM zeigt uns, ob wir fähig sind, es sinnvoll zu messen. [3]

3. Kosten und Nutzen von GQM

Um in einer Organisation ein Softwaremessprogramm zu etablieren, bedarf es natürlich nicht nur des praktischen Know-Hows, sondern, besonders für das Organisationsmanagement, ist auch der wirtschaftliche Aspekt eines solchen Programms von großer Bedeutung. In diesem Kapitel soll deshalb eine grobe Kosten-/Nutzenabschätzung für die Durchführung eines GQM Programms geliefert werden.

3.1. Kosten von GQM

Das hier präsentierte Kostenmodell ist dargestellt in [6] und basiert auf der Anwendung von sechs verschiedenen GQM Programmen.

Bei der Betrachtung der Kosten muss zwischen der regelmäßigen Anwendung und der erstmaligen Einführung eines GQM Programms unterschieden werden. Diese Unterscheidung ist notwendig, da bei der Ersteinführung eines Messprogramms ein verhältnismäßig höherer Aufwand besteht als bei der wiederholten Anwendung.

In jeder der im vorherigen Kapitel beschriebenen Phasen entstehen verschiedene Kosten, die hier kurz aufgeschlüsselt werden sollen. In der Planungsphase entstehen Kosten durch die Vorbereitung der Infrastruktur, die Festlegung von Verbesserungszielen, die Auswahl eines Projektes, die Erstellung eines Projektplans und durch die Ausbildung und Vorbereitung der Mitarbeiter, welche dann das GQM Team bilden. Dieses ist für die Durchführung bzw. für die weitere Planung des GQM Programms zuständig. In der Definitionsphase müssen die GQM „Goals", „Questions" und Metriken festgelegt werden, welche in einem GQM Plan zusammengefasst werden. Zudem führt man Interviews durch, welche die Definition von „Goals" unterstützen. Nach der Erstellung des GQM Plans muss, ausgehend von den identifizierten Metriken, ein Messplan erstellt werden. Schlußendlich hat noch die Simulation des Vorgehens zu erfolgen. In der Datensammlungsphase entstehen die Kosten durch die Einschulungen des Projektteams, welches die Mitarbeiter darstellt, die an dem zu messenden Projekt arbeiten. Daneben entstehen Kosten durch die tatsächliche Sammlung, Validierung, Kodierung und Speicherung der Daten. In der Interpretationsphase müssen die gesammelten Daten ausgewertet und für die Präsentation an die Zielgruppe vorbereitet werden.

Bei der Routine-Anwendung des GQM Programms stellt sich folgende Kostenstruktur dar [6]:

- Schätzungsweise 30% des Aufwandes wird für die Definierung des Messprogramms aufgewendet, während 70% für die tatsächliche Durchführung verwendet werden.
- 70 % des Aufwandes entfallen auf das GQM Team und nur 30% werden dem Projektteam zugerechnet.
- Der Aufwand des Projektteams für die Datensammlung beträgt weniger als 1% ihrer tatsächlichen Arbeitszeit.

Dagegen stellt sich bei der erstmaligen Anwendung eines GQM Programms folgende Kostenstruktur dar [6]:

- Der Gesamtaufwand für ein erstmalig durchgeführtes GQM Programm ist um ca. 60% höher als bei der Routine-Anwendung. Dies beruht darauf, dass sehr viel vorbereitende Arbeit durchgeführt werden muss (z.B. Training des GQM Teams).
- Schätzungsweise 50% des Aufwandes werden für die Definierung des Messprogramms aufgewendet. Im Vergleich dazu entfielen bei der Routine-Anwendung nur 30% auf diesen Posten. Vor allem die Erstellung des GQM Plans und des Messplans benötigt hierbei mehr Aufwand.
- 70% des Aufwandes werden vom GQM Team und 30% vom Projektteam verwendet, was keine Veränderung zur Routine-Anwendung darstellt.
- Der Aufwand des Projektteams ist weniger als 2% ihrer Gesamtarbeitszeit. Dies ist doppelt so viel wie in der Routine-Anwendung.

Der erhöhte Aufwand für die erstmalige Durchführung entsteht hauptsächlich durch zusätzliche Trainings- und Lernzeiten. Vor allem muss das GQM Team den Aufbau eines GQM Plans, und dabei die Definition von „Goals“ und daraus die Ableitung von „Questions“, erlernen.

Natürlich stellt die oben gezeigte Aufwandsverteilung nur Richtwerte dar, da einige Faktoren diese noch verschieben können. So erfordert etwa eine größere Anzahl von GQM „Goals“ auch einen erhöhten Aufwand bei allen Aktivitäten. Auch die Vergrößerung des Projektteams steigert den Aufwand für die Ausbildung und Datensammlung.

3.2. Nutzen von GQM

Der größte Nutzen eines Messprogramms liegt darin, die klar definierten produkt- oder prozessspezifischen Verbesserungsziele zu erreichen. Daneben kann eine Vielzahl anderer Vorteile aus der Verwendung von Messprogrammen gezogen werden [2] [4] [6].

GQM strukturiert die über Metriken gesammelten Daten und verbindet sie mit tatsächlich gestellten Fragen, während ohne die Verwendung von GQM unter Umständen unstrukturierte Daten gesammelt werden.

GQM unterstützt die Verbesserung von Arbeitsabläufen innerhalb einer Organisation, indem es fokussiert darstellt, wo Verbesserungen möglich sind. Daneben dokumentiert es, ob durchgeführte Veränderungen eine Verbesserung bringen.

Durch die direkte Einbeziehung der Mitarbeiter in den GQM Prozess, müssen sich diese mit den Qualitätssicherungmaßnahmen auseinandersetzen. Dies ermöglicht es ihnen, sich mit den Qualitätszielen stärker zu identifizieren.

Durch die verstärkte Zusammenarbeit der Mitarbeiter, die die Durchführung eines GQM Programms mit sich bringt, wird das Gruppengefüge verstärkt und die Kommunikation gesteigert.

Der finanzielle Vorteil an sich ist schwer zu messen, da eine starke Wechselwirkung zwischen den verschiedenen Einflussfaktoren besteht. Jedoch bringt eine Verbesserung der Prozessabläufe und Arbeitsmethoden oft auch eine höhere Produktivität der Mitarbeiter und damit reduzierte Kosten mit sich. Außerdem wirkt sich eine gute Qualität der Produkte positiv auf das Gesamtbild des Unternehmens aus, was zusätzliche Aufträge zur Folge haben kann.

4. GQM in der Praxis

4.1. Praxisbeispiel AT&T [3]

Das folgende Beispiel stammt von AT&T. Das Ziel der Entwickler war es, die Effektivität ihrer Software Inspektionen zu evaluieren. Die Hauptaufgabe war die Kosten/Nutzen Analyse der Inspektionen. Die GQM Methode wurde hierbei genutzt, um die zutreffendsten Metriken für die Analyse festzustellen, siehe Tabelle 2.

Tabelle 2 - AT&T Goals, Questions and Metriken

Goals	Questions	Metriken
Planung	Wie viel kostet der Inspektionsprozess?	Ø Aufwand pro KLOC % der Reinspektionen
	Wieviel Zeit benötigt der Inspektionsprozess?	Ø Aufwand pro KLOC Anzahl der KLOC die inspiziert wurden
Beobachtung und Kontrolle	Wie hoch ist die Qualität der inspizierten Software?	Ø Anzahl an Fehlern pro KLOC Ø Inspektionsrate Ø Vorbereitungsrate
	Mitarbeiterzuspruch zu den Prozeduren?	Ø Inspektionsrate Ø Vorbereitungsrate Ø LOC die inspiziert wurden % der Reinspektionen
	Wie ist der Status des Inspektions Prozesses?	Anzahl der KLOC die inspiziert wurden
Verbesserung	Wie effektiv ist der Inspektionsprozess?	Defektentfernungseffizienz Ø Anzahl an Fehlern pro KLOC Ø Inspektionsrate Ø Vorbereitungsrate

		Ø LOC die inspiziert wurden
	Wie ist die Produktivität des Inspektionsprozesses?	Ø Aufwand pro gefundenem Fehler
		Ø Inspektionsrate
		Ø Vorbereitungsrate
		Ø LOC die Inspiziert wurden

Nachdem die Entwickler die relevanten Metriken identifiziert hatten, gingen sie daran die Gleichungen und die Datenelemente, welche die Metriken beschreiben, zu spezifizieren. So stellt z.B. die Ø Vorbereitungsrate eine Funktion auf die Gesamtanzahl der inspizierten LOC, die Vorbereitungszeit für jede Inspektion und die Anzahl der Inspektoren dar. Für die Metrik wird ein Modell erstellt, welches sie als Gleichung darstellt. So wird etwa bei der Ø Vorbereitungsrate zuerst die Vorbereitungszeit für jede Inspektion durch die Anzahl der Inspektoren geteilt. Danach wird die Summe über alle Inspektionen gebildet und für die Normalisierung der Gesamtcodelänge verwendet.

Auch wenn die Metriken durch Gleichungen oder Verhältnisse beschrieben werden, ist die Definition nicht immer klar und eindeutig. Der GQM Plan liefert keine Aussage darüber, wie z.B. LOC zu messen sind, sondern stellt nur fest, dass irgendein Maß für die Codegröße erforderlich ist.

Die Resultate der Datensammlungsphase für 2 Projekte sind aus Tabelle 3 ersichtlich.

Tabelle 3 – AT&T Ergebnisse

Metrik	1. Projekt	2. Projekt
Anzahl der Inspektionen im Projekt	27	55
Anzahl der KLOC die inspiziert wurden	9,3	22,5
Ø LOC die inspiziert wurden	343	409
Ø Vorbereitungsrate	194 LOC/h	121,9 LOC/h
Ø Inspektionsrate	172 LOC/h	154,8 LOC/h
Ø Anzahl an Fehlern pro KLOC	106	89,7
% der Reinspektionen	11 %	0,5 %

Das Resultat für AT&T zeigte, dass zu schnell durchgeführte Inspektionen darin resultierten, dass weniger Fehler gefunden wurden.

Insgesamt kann gesagt werden, dass die Anwendung von GQM zur Selektion der relevanten Metriken für AT&T ein Erfolg war.

5. Conclusio

Die Einführung eines erfolgreichen Messprogramms in einer Organisation ist eine große Herausforderung. Die Gründe dafür sind zahlreich: verschiedene Sichtweisen, Modelle, Zwecke, Attribute und ihre Abhängigkeiten untereinander. GQM ist eine leistungsfähige Methode, die eine Datensammlung unterstützt, die genau auf die Ziele einer Organisation ausgerichtet ist. Diese Arbeit soll aufzeigen, dass der Einsatz von GQM in einer Organisation sinnvoll ist. Bei der Einführung eines GQM Programms darf jedoch nicht außer Acht gelassen werden, dass Schwierigkeiten auf verschiedenen Ebenen entstehen. Das Management muss überzeugt werden, dass ein solches Programm Vorteile bringt, und auch die Mitarbeiter müssen sensibilisiert und motiviert werden.

Wir sind der Meinung, dass GQM einen sehr sinnvollen Ansatz zur zielorientierten Softwaremessung darstellt. Die Grundstruktur von GQM, wie sie in dieser Arbeit dargelegt wurde, beinhaltet jedoch sicherlich nicht alle Aspekte die sich ein Unternehmen von einer solchen Methode erhofft. Toolunterstützung und eine verfeinerte Methode zur Definierung der „Questions“ sind nur Beispiele dafür, siehe auch [2].

6. Referenzen

[1] Basili, V.R., G. Caldiera, and H.D. Rombach, “Goal Question Metric Paradigm” in *Enciclopedia of Software Engineering* (J.J. Marciniak, ed.), Vol. 1, John Wiley & Sons, 1994, pp. 528-532

[2] A. Birk, R. van Solingen, and J. Järvinen,”Business Impact, Benefit, and Cost of Applying GQM in Industry: An In-Depth, Long-Term Investigation at Schlumberger RPS”, *Proceedings of the 5th International Symposium on Software Metrics (Metrics ‘98)*, Bethesda Maryland, November 1998

[3] Fenton, N.E., and S.L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach* Second Edition, International Thomson Publishing Inc., London u.a., 1996

[4] A. Fuggetta, L. Lavazza, S. Morasca, S. Cinti, G. Oldano, and E. Orazi, “Applying GQM in an Industrial Software Factory”, *Transactions on Software Engineering and Methodology*, Vol. 7, No. 4, ACM, October 1998, pp. 411-448

[5] S.L. Pfleeger, and C.L. McGowan, “Software Metrics in process maturity framework”, *Journal of Systems and Software*, 27(9), ELSEVIER, 1990, pp. 255-261

[6] Solingen, R.v., and E. Berghout, *The Goal/Question/Metric Method: a practical guide for quality improvement of software development*, McGraw-Hill Publishing Company, London, 1999

Objektorientierte Softwaremetriken

Johannes Wernig-Pichler
9961408
jwernigp@edu.uni-klu.ac.at

Mario Lassnig
9961428
mario.lassnig@uni-klu.ac.at

ZUSAMMENFASSUNG

Mit dem Aufkommen der objektorientierten Programmierung konnten viele der bis dahin entwickelten Metriken zwecks der Besonderheiten, welche dieses neue Programmierparadigma mit sich brachte, nicht mehr auf die damit entwickelte Software angewendet werden. Somit war man gezwungen, neue passende Metriken zu entwickeln, welche die Besonderheiten der Objektorientierung berücksichtigen. In dieser Arbeit werden die Grundlagen von objektorientierten Metriken vorgestellt und auf die Unterschiede zu prozeduralen Metriken eingegangen. Weiters werden objekt-orientierte Softwaremetriken vorgestellt, beginnend mit den Metriken welche von Chidamber und Kemerer entworfen wurden und die zu den meistreferenzierten Metriken auf diesem Gebiet gehören. Abgeschlossen wir die Arbeit durch das Aufzeigen zweier Problemstellungen, der Grenzwertproblematik und dem Einfluss der Klassengröße auf die Validierung von Metriken.

1. EINLEITUNG

In der heutigen Geschäftswelt sind Softwarefirmen aufgrund der starken Konkurrenz gezwungen verlässliche Software in immer kürzeren Releasezyklen zu entwickeln. Dies gilt vor allem für Bereiche in denen höchst verlässliche Software benötigt wird, wie zum Beispiel in der Telekommunikation oder der Luft- und Raumfahrt. Aufgrund der hohen Anforderungen an die Software wird für diese ein entsprechendes Qualitätsmanagement benötigt mit dem Attribute wie Testbarkeit, Zuverlässigkeit, Fehleranfälligkeit oder Wartbarkeit messbar sind. Da jedoch viele dieser Attribute erst spät im Entwicklungsprozess direkt gemessen werden können, ist es wichtig Softwaremetriken zu haben, die schon frühzeitig als Indikatoren für diese Attribute dienen können. Softwarefirmen können Softwaremetriken auf zumindest drei verschiedene Arten einsetzen [9]: um Vorhersagen über das System zu treffen, um frühzeitig Risikokomponenten zu identifizieren und zur Bestimmung von Design- & Programmierrichtlinien. Dies erlaubt es Unternehmen z.B. eine frühzeitige Einschätzung der Systemqualität zu treffen und Aktionen zu ergreifen, um die Anzahl der fehlerhaften Softwarekomponenten zu verringern.

1.1 Vorhersagen über das System

Normalerweise werden Softwaremetriken für einzelne Komponenten eines einzelnen Systems gesammelt. Vorhersagen über die Einzelkomponenten können jedoch vereinigt werden, um eine Vorhersage über das gesamte System zu ermöglichen. So können etwa Voraussagen über die Fehleranfälligkeit jeder Klasse des Systems benutzt werden, um daraus Schlüsse über die allgemeine Qualität des Systems zu ziehen.

1.2 Identifizierung von Risikokomponenten

Die Definition einer Risikokomponente variiert abhängig vom verwendeten Kontext. Mögliche Definitionen sind [9]:

- eine Risikokomponente ist eine, die irgendwelche Fehler enthält, die während des Testens gefunden werden
- eine Risikokomponente ist eine, deren Korrektur nach dem Auffinden eines Fehlers äußerst kostenintensiv ist

Wenn diese Komponenten frühzeitig identifiziert werden, ist das Unternehmen in der Lage vorbeugende Aktionen durchzuführen. Dies können etwa die Konzentration von Fehlerfindungsaktivitäten auf Risikokomponenten sein, das Neudesignen von jenen Komponenten, die höchstwahrscheinlich Fehler verursachen werden oder kostenintensiv zu warten sind, oder die entsprechende (je nach Risiko) Zuteilung von Testressourcen.

1.3 Design- und Programmierrichtlinien

Mit der Hilfe von Softwaremetriken kann ein Unternehmen Richtlinien für das Design und die Programmierung erlassen. Hierbei werden unter anderem Grenzwerte für die einzelnen Metriken definiert, die akzeptable von nicht akzeptablen Werten abgrenzen. Softwaremetriken können also dazu verwendet werden um schlecht geschriebenen Programmcode oder schlechtes Programmdesign aufzuzeigen. Aber auch für Manager sind solche Grenzwerte von Nutzen, da extreme Wertabweichungen signalisieren, dass möglicherweise Maßnahmen seitens des Managements nötig sind.

2. GRUNDLAGEN FÜR METRIKEN DER OBJEKTORIENTIERUNG

Die Grundlagen von objektorientierter Softwaremetriken basieren auf den Besonderheiten der Struktur von objektorientierten Softwaresystemen. Da sich diese extrem von prozeduralen Merkmalen unterscheiden, seien hier nun die wichtigsten Merkmale erwähnt und beschrieben.

2.1 Kopplung

> Die Kopplung einer Einheit beschreibt die Wechselbeziehungen, die sie zu anderen Einheiten unterhält und die damit verbundenen Abhängigkeiten.[14]

Kommunizieren zwei Objekte mittels Nachrichtenaustauschs miteinander, wobei diese Objekte nicht in einer Hierarchiebeziehung zueinander stehen, spricht man von Kopplung. Je höher der Nachrichtenfluss zwischen den Objekten, desto undurchsichtiger werden Funktionalität und Wirkungsweise der beteiligten Objekte. Diese Interaktivität erhöht die Kopplung im System und erschwert Modularität, Reuse und Erweiterbarkeit. Insbesondere gibt die Kopplung Hinweise auf die Sensibilität einer Einheit im Hinblick auf Änderungen in anderen Teilen des Systems.

2.2 Vererbung

Je tiefer eine Klasse in der Vererbungshierarchie liegt, desto mehr Methoden erbt diese Klasse und desto mehr Überschreibungen und Überladungen der Methoden sind möglich. Änderungen in Superklassen führen zu direkten Änderungen in allen Subklassen. Alle eventuellen Auswirkungen dieser Änderungen müssen beachtet werden, um die Komplexität des Systems bestimmen zu können. Ziel ist es, eine ausgewogene Vererbungshierarchie zu finden, welche weder zu einer zu schwachen noch einer zu starken Vererbung tendiert.

2.3 Kohäsion

> Die Bindung (Synonym: Kohäsion) beschreibt den Grad der Zusammengehörigkeit der zu einer Einheit zusammengefassten Elemente.[14]

Softwaresysteme die objektorientiert entwickelt werden, sind in ihrer Gesamtheit als unabhängige, miteinander kommunizierende Einheiten anzusehen. Die Bildung der verschiedenen Einheiten sollte möglichst nachvollziehbar sein und klar definierten inhaltlichen Konzepten folgen.

2.4 Volumen

> Das Volumen eines Softwareprodukts wird bestimmt durch die Anzahl der Informationen, die für seine korrekte Erstellung, Nutzung, Wartung, Erweiterung, Änderung oder Verwaltung verarbeitet und verstanden werden muß.[14]

Im Normalfall sind große Objekte schwieriger und komplizierter zu verstehen als kleinere. Große Objekte können oft durch Zerlegen in mehrere kleine, in sich geschlossene Einheiten an Komplexität verlieren. Da große Objekte oft speziell auf eine Applikation angepasst sind, können sie kaum in anderen Programmen wiederverwendet werden.

2.5 Kapselung

Unter Kapselung versteht man die Abgrenzung von Daten und Methoden vor externem Zugriff. Wird Information benötigt, so werden diese über wohldefinierte Schnittstellen bereitgestellt, wobei die dahinterliegenden Daten und Methoden nach außen hin nicht sichtbar sind. Programmierer können somit nur über das Interface eines Objektes auf das Objekt zugreifen, womit die Implementierung des Objektes nach außen hin nicht sichtbar ist.

2.6 Unterschiede zu Metriken für prozedurale Softwaresysteme

Die Unterschiede zwischen prozeduralen und objektorientierten Metriken legen sich schon im grundlegenden Design der zwei Paradigmen fest. So werden in der objektorientierten Technologie Objekte, und nicht Algorithmen, als die fundamentalen Baublöcke von Software verwendet. Weiters zeichnet sich die objektorientierte Softwareentwicklung vor allem durch die Anwendung neuer Strukturierungsmerkmale gegenüber der prozeduralen Softwareentwicklung aus. Solch Strukturierungsmerkmale sind etwa: Kapselung, Abstraktion, Klassenhierarchie (Vererbung), Polymorphie und der Austausch von Nachrichten. Ein weiterer wichtiger Unterscheidungspunkt ist der Einsatz von umfangreichen Klassenbibliotheken, die den Reuse von Software unterstützen sollen. Aufgrund dieser Besonderheiten der Struktur von objektorientierter Software konnten viele der traditionellen Metriken nicht mehr angewendet werden. Dies vor allem wegen der Unfähigkeit der Metriken, Konzepte wie Vererbung, Polymorphismus und Nachrichtenaustausch zu erfassen und zu bewerten. Darüber hinaus konnten Softwaremetriken aus anderen Programmierparadigmen wie zum Beispiel die Anzahl der Programmzeilen oder die Länge des Programms die Komplexität von objektorientierter Software nur unzureichend beurteilen. Für weitere Beschreibungen der Eigenschaften von objektorientierten Softwaremetriken siehe [18].

3. DIE METRIKEN VON CHIDAMBER UND KEMERER

1994 definierten Shyam Chidamber und Chris Kemerer in [5] eine große Menge von objektorientierten Softwaremetriken, wobei mittels sechs Kennzahlen die Qualität eines objektorientierten Systems beschrieben wird. Innerhalb der Welt objektorientierter Softwaremetriken gehören die Metriken von Chidamber und Kemerer zu den Meistreferenzierten. Im folgenden werden diese Metriken vorgestellt.

3.1 Weighted Methods per Class ... WMC

DEFINITION 1. *Gegeben sei eine Klasse C mit Methoden $M_1 \dots M_n$, welche in der Klasse C definiert sind. Seien $c_1 \dots c_n$ die Komplexität der Methoden. Dann gilt:*

$$WMC = \sum_{i=1}^{n} c_i$$

WMC ist somit die Summe der Komplexitäten aller Methoden M der Klasse C. Sind die Komplexitäten der Methoden ident, dann gilt WMC $= n$, äquivalent zur Anzahl der Methoden der Klasse.

Die Anzahl und Komplexität der Methoden beeinflussen, wieviel Zeit und Aufwand benötigt werden um eine Klasse zu entwickeln und zu warten. Je größer die Anzahl der Methoden, desto größer ist ein möglicher Einfluss auf Subklassen, da Subklassen alle Methoden der Superklasse erben.

Klassen mit einer großen Anzahl an Methoden sind meist anwendungsspezifisch und daher nicht besonders für Reuse geeignet.

3.2 Depth of Inheritance Tree . . . DIT

DEFINITION 2. *Die Metrik gibt die Tiefe der Klasse C von der Wurzel des Vererbungsbaumes an. Bei Mehrfachvererbung gibt die Metrik die maximale Tiefe im Baum an.*

Je tiefer eine Klasse in der Hierarchie steht, umso größer ist die Wahrscheinlichkeit dass diese Klasse eine größere Anzahl an Methoden erbt, wodurch ihr Verhalten schwieriger vorauszusehen ist. Große Komplexität im Design führt zu einem tieferen Vererbungsbaum, da viel mehr Methoden und Klassen beteiligt sind. Die Wahrscheinlichkeit, dass eine ererbte Methode einer Superklasse verwendet wird, ist in tieferen Klassen höher.

3.3 Number of Children . . . NOC

DEFINITION 3. *Die Metrik gibt die Anzahl der unmittelbaren Subklassen der Klasse im Vererbungsbaum an.*

Die Auswirkungen einer Klasse mit vielen Subklassen auf das Gesamtsystem kann mit dieser Metrik abgeschätzt werden, da Vererbung auch eine Form von Reuse darstellt. Gibt es viele Subklassen, ist die Wahrscheinlichkeit hoch, dass Superklassen ungenau designed wurden, was einen möglichen Missbrauch von Subklassen darstellen kann. Ebenfalls gibt die Anzahl der Subklassen an, welchen Gesamteinfluss diese Klasse auf das System bildet. Bei einer Modifikation einer Klasse mit hoher Subklassenanzahl wird dadurch die Notwendigkeit des Testens anderer Komponenten essentiell.

3.4 Coupling between Object Classes . . . CBO

DEFINITION 4. *Die Metrik gibt die Anzahl der Klassen an, mit denen eine Klasse in einer Kopplungsbeziehung steht.*

Übermäßige Kopplung zwischen Klassen steht im Gegensatz zu modularem Design und verhindert Reuse. Je unabhängiger eine Klasse ist, desto einfacher ist es, sie in anderen Systemen zu verwenden. Um Modularität und Kapselung zu verbessern, sollte die Kopplung zwischen Klassen auf ein Minimum reduziert werden. Ist die Kopplung zwischen Klassen zu groß, ist die Anfälligkeit anderer Klassen auf Änderungen im Design ebenfalls hoch und erschwert das Warten. Eine Kopplungsmetrik ist somit nützlich um zu bestimmen, wie komplex der Testprozess für das Design von einzelnen Komponenten sein könnte. Je größer die Kopplung, desto ausführlicher muss getestet werden.

3.5 Response for a Class . . . RFC

DEFINITION 5. *$RFC = |RS|$, wobei das RS (Response Set) einer Klasse die Menge aller lokalen Methoden und die durch lokale Methoden der Klasse gerufenen Methoden ist.*

Diese Metrik ist somit die Anzahl aller verschiedenen Methoden, die in der untersuchten Klasse aufgerufen werden, zuzüglich der Methoden, welche die Klasse selbst besitzt. RFC zeigt somit die theoretisch mögliche Anzahl an Methoden an, die durch Senden einer Nachricht an ein Objekt ausgelöst werden können. Je größer die Anzahl ist, desto komplexer ist die Klasse programmiert. Weiters wird durch einen hohen RFC-Wert vor allem das Testen der Methoden und das Finden von Fehlern in den Programmen entsprechend aufwendiger, da vom Tester ein größeres Verständnis benötigt wird.

3.6 Lack of Cohesion in Methods . . . LCOM

DEFINITION 6. *Sei C_1 eine Klasse mit n Methoden M_1, M_2, . . . , M_n. Sei $\{I_j\}$ = die Menge aller Instanzvariablen, die von der Methode M_j genutzt wird. Dann gibt es n solche Mengen $\{I_1\}$, ... $\{I_n\}$. Sei $P = \{(I_i, I_j) | I_i \cap I_j = \emptyset\}$ und $Q = \{(I_i, I_j) | I_i \cap I_j \neq \emptyset\}$. Wenn alle n Mengen $\{I_1\}$, ... $\{I_n\}$ $\emptyset$ sind, dann sei $P = \emptyset$.*

$$LCOM = \begin{cases} |P| - |Q| & wenn\ |P| > |Q| \\ 0 & sonst \end{cases}$$

Um den Mangel an Kohäsion in einer Klasse zu messen, sortiert man die Methoden einer Klasse nach dem Satz von Variablen, die sie benutzen oder verändern. LCOM ist nun die Anzahl der verschiedenen genutzten Variablensätze. Benutzen alle Methoden die gleichen Variablen, so existiert nur ein Variablensatz und LCOM hat den Wert eins. Ein Mangel an Kohäsion deutet darauf hin, dass Klassen möglicherweise in zwei oder mehrere Subklassen aufgeteilt gehören. Geringe Kohäsion erhöht weiters die Komplexität, womit auch die Wahrscheinlichkeit von Fehlern während des Entwicklungsprozesses steigt.

4. ANDERE METRIKEN

Aus der Menge der inzwischen über 200 definierten Metriken für objektorientierte Software werden folgend einige weitere Metriken angeführt, darunter Metriken von Sharble und Cohen sowie Lorenz und Kidd.

4.1 Metriken von Sharble und Cohen

Bei ihrer Untersuchung von Vorgehensweisen zur Entwicklung von objektorientierten Programmen benutzten Sharble und Cohen die Metriken von Chidamber und Kemerer und erweiterten diese um drei Metriken: Weighted attributes per class, Number of tramps, Violations of the Law of Demeter (siehe: [7]).

4.1.1 Weighted attributes per class $\cdots$ WAC

Das Ergebnis dieser Metrik ist die Anzahl der Variablen einer Klasse, gewichtet mit dem Umfang der Klasse. Auch hier werden, analog zur Metrik WMC, nur die Variablen gemessen und nicht die Methoden. Das Ergebnis von WAC ist ein Indiz dafür, wieviel Zeit und Aufwand erforderlich ist, das Objekt weiterzuentwickeln und zu warten. Je mehr Variablen in einer Klasse vorhanden sind, desto größer ist die Auswirkung auf die Unterklassen, da diese alle Variablen erben.

4.1.2 Number of tramps $\cdots$ NOT

Die Bezeichnung von Methoden, also deren Namen, und alle Parameter, mit denen sie aufgerufen werden, stellen einen Hinweis auf die Komplexität der Methoden dar, da eine Methode mit vielen Parameter im allgemeinen schwieriger zu durchschauen ist als eine einfache Methode. Die Metrik *Anzahl der Methodenparameter* ist als die Gesamtanzahl von allen Parametern aller Methoden der Klasse definiert.

4.1.3 Violations of the Law of Demeter ··· VOD

Mit der Metrik *Verstöße gegen das Gesetz von Demeter* hat die Aussage des Gesetzes von Demeter als Grundlage. Das Gesetz von Demeter beschränkt die Nachrichtenverbindungen zwischen Klassen, d.h. es versucht die Kopplung zu begrenzen. Nachrichten eines Objektes sind nur erlaubt an: sich selbst (this); Parameterobjekte; Objekte, die das Objekt erzeugt; Komponentenobjekte. Das Grundprinzip dahinter ist, dass keine Nachrichten an Objekte geschickt werden sollen, die Rückgabewerte anderer Operationen sind. (siehe: [13]). Das Ergebnis der Metrik ist die Anzahl der Methoden in der Klasse, die gegen dieses Gesetz verstoßen.

4.2 Metriken von Lorenz & Kidd

In [15] stellen Lorenz und Kidd eine Menge objektorientierter Metriken vor. Diese beziehen sich auf die Methodengröße, die Klassengröße, die Klassenvererbung, die Methodenvererbung, die internen Merkmale von Methoden, die internen Merkmale von Klassen und die externen Merkmale von Klassen. Im folgenden wird zu jedem dieser Bereiche zumindest eine Metrik aufgeführt.

4.2.1 Methodengrösse

Eine Metrik, die sich auf die Methodengröße bezieht, ist *Number of message sends*, welche die Anzahl von in der Methode gesendeten Nachrichten misst. Dabei können die Nachrichten die Typen Unary (Das sind Nachrichten ohne Argumente.), Binary (Nachrichten mit einem Argument, getrennt durch einen speziellen Selektor. Beispiele sind Stringkonkatenation und mathematische Funktionen.) oder Keyword (Nachrichten mit mehr als einem Argument.) haben. Diese Metrik soll die Klassengröße auf eine relativ unbefangene Art messen. Eine große Anzahl von Nachrichten kann auf funktionsorientierte Programmierung hindeuten. Der Grenzwert für diese Metrik wird mit dem Wert 9 angegeben.

Number of statements ist eine weitere Metrik, die die Methodengröße misst. Was als Anweisung bezeichnet wird, hängt natürlich von der Programmiersprache ab. Als Grenzwert wird der 0.8-fache Wert vom *Number of message sends* - Grenzwert angeführt.

4.2.2 Klassengrösse

Metriken wie *Number of instance methods in a class*, *Number of instance variables in a class* oder *Number of class variables in a class* werden dazu verwendet einzelne Klassen zu beurteilen. Die Metrik *Number of instance methods in a class* zählt alle öffentlichen, geschützten und privaten Methoden, die für eine Klasse definiert sind. Damit soll der Grad der Zusammenarbeit zwischen den Klassen gemessen werden. Als Grenzwert wird 20 vorgeschlagen.

4.2.3 Klassenvererbung

Abstrakte Klassen existieren um Reuse von Methoden zu ermöglichen und Daten unter ihren Subklassen festzulegen. Die Anzahl von abstrakten Klassen im System ist ein Anzeichen für den erfolgreichen Gebrauch von Vererbung. *Number of abstract classes* zählt die Anzahl dieser Klassen im System. Der Grenzwert wird mit zehn bis fünfzehn Prozent der Gesamtanzahl an Klassen im System angegeben.

4.2.4 Methodenvererbung

Diese Gruppe von Metriken untersucht die Superklassen – Subklassen Vererbungsbeziehungen. Die Metrik *Number of methods inherited by a subclass* soll hierbei ein Indikator für die Stärke des Einsatzes des Subklassenkonzepts sein. Lorenz und Kidd legen für diese Metrik keinen Grenzwert fest, geben aber an, dass der Prozensatz an vererbten Methoden hoch sein sollte.
Die allgemeine Funktion einer Subklasse, nämlich dass es den Objekttyp der Superklasse weiter spezialisiert, wird von der Metrik *Number of methods added by a subclass* wiedergespiegelt. Die Anzahl neuer Methoden sollte dabei normalerweise fallen, je tiefer hinunter in der Hierarchie man geht. Als unterer Grenzwert wird 1 angegeben. Der obere Grenzwert stellt eine absteigende Skala dar. Für den ersten Level der Verschachtelung wird ein Wert von 20 angeführt, für den Level sechs, welcher der höchste Verschachtelungslevel sein sollte, ein Wert von kleiner gleich vier.

4.2.5 Interne Merkmale von Methoden

Diese Gruppe von Metriken beschäftigt sich mit den internen Charakteristiken von Methoden. Mittels der Metrik *Method complexity* soll die Komplexität von Methoden bestimmt werden. Hierbei werden den Konstrukten der Programmiersprache (API Aufrufe, Zuweisungen, binäre Ausdrücke, arithmetische Operatoren, Nachrichten, Parameter, geschachtelte Ausdrücke, temporäre Variablen, unäre Ausdrücke usw.) Werte zugewiesen, die dann summiert werden. Als Grenzwert empfehlen Lorenz und Kidd den Wert 65.

4.2.6 Interne Merkmale von Klassen

Diese Metriken beziehen sich auf das Design der internen Charakteristiken von Klassen, etwa wie Instanzvariablen verwendet werden oder welche externen Referenzen sie anlegen. Die Metrik *Global Usage* zählt die Anzahl von globalen Variablen, wie Systemvariablen oder Klassenvariablen. Da ein vermehrter Einsatz von globalen Variablen auf schlechtes objekt-orientiertes Design hindeutet, sollte deren Einsatz reduziert werden. Als Grenzwert wird eine Systemvariable angegeben, jeder weitere Einsatz von globalen Variablen muß gut begründet werden.

4.2.7 Externe Merkmale von Klassen

Metriken dieser Art untersuchen, wie sich eine Klasse auf andere Klassen, Subsysteme, Benutzer usw. bezieht. Die Metrik *Number of times a class is reused* bezieht sich auf einen der großen Vorteile der Objektorientierung, die zusätzliche Unterstützung von Reuse. Auch hier wird kein Grenzwert angegeben. Vielmehr wird darauf hingewiesen, dass jedes Projekt eine Menge von Klassen haben soll, die später wiederbenützt werden können und dass jedes Projekt eine Menge von Klassen wiederbenützen sollte.

4.3 Weitere Metriken

In [19], [2], [17], [16] und [6] findet man weitere Auflistungen von objektorientierten Softwaremetriken. Unter diesen Metriken ist den Autoren die Metrik VLT (Volatility) [16] besonders aufgefallen, da sie auf Wahrscheinlichkeit beruht. Die Metrik stellt hierbei ein Wahrscheinlichkeitsmaß für die

Änderung eines Objekts dar, wobei die zugrundliegenden Wahrscheinlichkeiten vom Designer subjektiv eingeschätzt werden müssen.
Weitere interessante Metriken sind zum Beispiel MEF (Maintainability Efficiency) sowie RUF (Reuse Frequency), welche die Wartbarkeitseffizienz von Methoden bzw. die Frequenz der Wiederverwendung im System messen (siehe: [19]).

5. OBJEKTORIENTIERTE METRIKEN & GRENZWERTE

Bei der Verwendung von Metriken und der Interpretation der Ergebnisse ist folgendes zu beachten:

> *Wann* gibt ein Messwert Anlass für Gegenmaßnahmen? Gesucht ist also eine Funktion, welche die Menge der Messwerte in auffällige und unauffällige Werte aufteilt.[12]

Gesucht sind also (Grenz)Werte, die für die Interpretation von Messergebnissen verwendet werden können und die dabei helfen Probleme im System aufzuspüren. Der Begriff Grenzwert kann dabei folgend definiert werden:

> Thresholds are heuristic values used to set ranges of desirable and undesirable metric values for measured software. These thresholds are used to identify anomalies, which may or may not be an actual problem.[15]

Das heißt, wenn für eine Metrik ein solcher Grenzwerteffekt identifiziert wäre, dann könnten Klassen mit Werten die unterhalb dieses Grenzwertes liegen als sicherer Bereich angesehen werden, und Programmierer, die ihre Klassen bewusst auf diesen Bereich beschränken, hätten Gewissheit, dass in diesen Klassen kaum Probleme bzw. Fehler auftreten. Ein solcher Grenzwert wird in Abbildung 1 [3] auf der rechten Seite dargestellt. Da für die meisten Metriken jedoch kein solcher Grenzwert bekannt ist, wurden Grenzwerte aufgrund von empirischen Wissen, also Erfahrung, festgelegt (wie etwa in [15]). Hierbei wird bei einem gewissen Wert eine Grenzlinie gezogen und Klassen, deren Werte über diesem Grenzwert liegen, werden die sein, die am meisten Probleme bzw. Fehler aufweisen. Somit können solche Grenzwerte zwar für die Identifizierung der fehleranfälligsten Klassen verwendet werden, jedoch können Klassen mit Werten unterhalb dieses Grenzwertes trotzdem eine hohe Fehlerrate aufweisen, jedoch nicht die höchste. Ein solcher Grenzwert wird in Abbildung 1 [3] auf der linken Seite dargestellt.

Natürlich wären erstere Grenzwerte für Programmierer von größerem Vorteil als Letztere, aber im allgemeinen scheint es schwierig zu sein solche Grenzwerte zu finden. Manche Forscher (etwa [11]) meinen sogar, dass erstgenannte Grenzwerte für bis heute definierte objektorientierte Metriken gar nicht existieren. Und da die Entwicklung von Software auch immer eine menschliche Komponente enthält (etwa Systemdesigner, Programmierer), und die beteiligten Personen unterschiedliche Erfahrungen und Wissen besitzen, scheint es in der Tat fraglich, ob solch allgemein gültige Grenzwerte existieren. Viel mehr scheint es, als ob es anstatt fixer Grenzwerte Grenzbereiche gibt, die eine Art von Richtlinien für die an der Systementwicklung beteiligten Personen darstellen[1].

[1] Was der Idee empirisch festgelegter Grenzwerten entspricht.

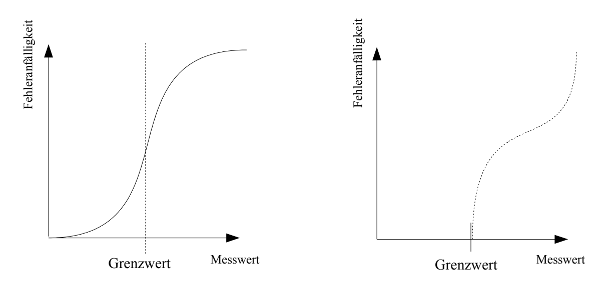


Abbildung 1. - Verschiedene Typen von Grenzwerten

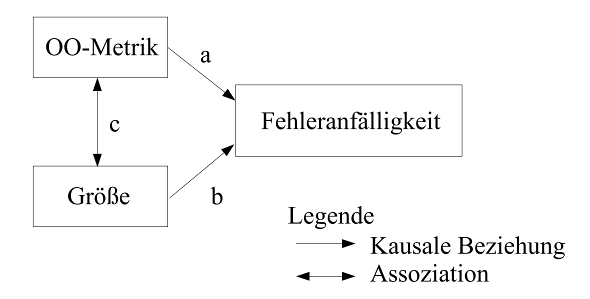


Abbildung 2. - Pfaddiagramm, dass den Verzerreffekt der Klassengröße auf objektorientierte Metriken illustriert

6. EINFLUSS DER KLASSENGRÖSSE AUF METRIKEN

Mit dem Aufkommen objektorientierter Softwaremetriken wurde es nötig diese sowohl theoretisch als auch empirisch zu validieren. Obwohl mehrere Metriken überprüft wurden und ein Großteil der Literatur zur Validierung (etwa [1] bezüglich der Metriken von Chidamber und Kemerer) von objektorientierten Metriken diese als validiert betrachtet, haben viele der Validierungsstudien den potenziellen Verzerrungseffekt[2] der Klassengröße auf die einzelnen Metriken ignoriert [10].
Das kommt daher, dass viele der durchgeführten Analysen eindimensional sind: sie modellieren nur die Beziehung zwischen der Metrik und der abhängigen Variable. So wurde etwa in [4] mit Hilfe von eindimensionalen Modellen die Beziehung zwischen der Fehleranfälligkeit von Klassen und objektorientierten Metriken, sowie die Möglichkeit wie etwa Metriken eingesetzt werden können um vorherzusagen, wo Fehler vorhanden sind, untersucht. Abbildung 2 (siehe: [8]) zeigt den Verzerreffekt der Größe auf die Fehleranfälligkeit. Der Pfad a) repräsentiert die weit verbreitete Annahme, dass Metriken ein Bezugselement für die Fehleranfälligkeit sind. Pfad b) zeigt einen positiven ursächlichen Zusammenhang zwischen Größe und Fehleranfälligkeit. Der Pfad c) stellt eine positive Assoziation zwischen Metriken und Größe dar. Wenn diese Graphik der Realität entspricht, dann verzerrt die Größe die Beziehung zwischen Metriken und Fehleranfäl-

[2] Größere Klassen führen automatisch zu einer höheren Fehlerwahrscheinlichkeit.

ligkeit, und da die Größe ein positiver Verzerrer ist, werden die Assoziationen immer positiver dargestellt, als sie wirklich sind.

In [10] und [8] wurden Analysen[3] bezüglich der Auswirkung des Verzerreffektes der Größe auf bereits validierte objektorientierte Metriken durchgeführt. In dieser Studie wurden fünf Metriken aus [5] und zwei Metriken aus [15] verwendet. Die Autoren kamen zu dem Ergebnis, dass die Größe einen starken Einfluss auf WMC, DIT, RFC, LCOM und NMA[4] hat. Nachdem die Verzerreffekte der Größe korrigiert wurden, wiesen die Metriken nur mehr einen sehr schwachen (WMC, RFC, NMA) oder gar keinen Zusammenhang (DIT, LCOM, NPAVG[5]) mit der Variable Fehleranfälligkeit auf. Aufgrund dieser Ergebnisse scheint es ratsam zu sein, vorhergehende Validierungsstudien nochmals kritisch zu hinterfragen und den Verzerreffekt der Größe entsprechend zu berücksichtigen.

7. CONCLUSIO

Auch wenn auf dem Gebiet der objektorientierten Softwaremetriken schon über zehn Jahre Forschung betrieben wird und inzwischen über 200 Metriken vorgeschlagen wurden, ist es im Vergleich zum Gebiet der traditionellen Metriken, dessen Wurzeln in die sechziger und siebziger Jahre zurückgehen, noch relativ jung. Somit ist es nicht verwunderlich, dass es noch einige Probleme und Fragestellungen (im Besonderen die Grenzwertproblematik und der Einfluß der Klassengröße auf die Validierung von Metriken) gibt, denen noch vermehrt Aufmerksamkeit geschenkt werden muss. Und auch wenn den Anwendern dieser objektorientierten Metriken eine große Menge von Metriken zur Verfügung steht, stellt die Auswahl der für sie passenden Metriken oft ein großes Problem dar. Es wäre daher von Vorteil, wenn es einen Kernsatz von ausreichend validierten Metriken geben würde, mit denen die wesentlichen Aspekte von objektorientierter Software verlässlich beurteilt werden könnten. Die Definition eines solchen Kernsatzes von Metriken sollte nach Meinung der Autoren in naher Zukunft ein wichtiges Ziel der Forschung auf diesem Gebiet sein.

8. LITERATUR

[1] V. Basili, L. Briand, W. Melo, *A Validation of Object-Oriented Design Metrics as Quality Inidcators*, IEEE Transactions on Software Engineering, Volume 22, No. 10, pp. 751-761, Oct. 1996

[2] D. Bellin, M. Tyagi, M. Tyler, *Object-Oriented Metrics: An Overview*, Web Publication, 1999, URL: http://www.cs.ubc.ca/local/reading/proceedings/cascon94/papers/bellin.ps

[3] S. Benlarbi, K. El Emam, N. Goel, S. Rai, *Thresholds for Object-Oriented Measures*, NRC/ERB-1073, NRC Publication Number: NRC 43652, March 2000, URL: http://iit-iti.nrc-cnrc.gc.ca/iit-publications-iti/docs/NRC-43652.pdf

[3] Die Analysen wurden nur für die Variable Fehleranfälligkeit durchgeführt und gelten somit nur für diese und nicht für andere Variablen, wie etwa Wartbarkeit und Wiederverwendbarkeit.

[4] Number of methods added by a subclass [15]

[5] Average numbers of parameters per method [15]

[4] L. Briand, J. Wüst, J. Daly, V. Porter, *Exploring the Relationships between Design Measures and Software Quality in Object-Oriented Systems*, Journal of Systems and Software, Volume 51, Issue 3, pp. 245-273, 2000

[5] S. Chidamber, Ch. Kemerer, *A Metrics Suite for Object Oriented Design*, IEEE Transactions on Software Engineering, Volume 20, No. 6, June 1994, pp. 476 - 493

[6] R. Dumke, E. Foltin, *Metrics-based Evaluation of Object-Oriented Software Development Methods*, Preprint Nr. 10, Fakultät für Informatik, Universität Magdeburg, 1997, URL: http://ivs.cs.uni-magdeburg.de/sw-eng/agruppe/forschung/paper/ooeval.pdf

[7] Ch. Ebert, R. Dumke, *Software-Metriken in der Praxis: Einführung und Anwendung von Software-Metriken in der industriellen Praxis*, Springer Verlag, 1996.

[8] K. El Emam, S. Benlarbi, N. Goel, S. Rai, *A Validation of Object-oriented Metrics*, NRC/ERB-1063, NRC Publication Number: NRC 43607, October 1999, URL: http://iit-iti.nrc-cnrc.gc.ca/publications/nrc-43607_ e.html

[9] K. El Emam, *A Methodology for Validating Software Product Metrics*, NRC/ERB-1076, NRC Publication Number: NRC 44142, June 2000, URL: http://iit-iti.nrc-cnrc.gc.ca/iit-publications-iti/docs/NRC-44142.pdf

[10] K. El Emam, S. Benlarbi, N. Goel, S. Rai, *The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics*, IEEE Transactions on Software Engineering, Volume 27, No. 7, pp. 630-650, July 2001

[11] K. El Emam, *Object-Oriented Metrics: A Review of Theory and Practice*, Advances in software engineering, Springer-Verlag New York, Inc., New York, NY, 2002

[12] K. Erni, *Anwendung multipler Metriken bei der Entwicklung objektorientierter Frameworks*, 5th Workshop on Software Metrics, Berlin, 1995, URL: http://irb.cs.tu-berlin.de/z̃use/gi/erni.ps.gz

[13] B. Kahlbrandt, *Skript zur Vorlesung Software-Engineering II*, URL: http://users.informatik.haw-hamburg.de/˜khb/st4se2/node146.html, 2001

[14] K. Kuhlmann, *Komplexität, Kopplung und Bindung bei der objektorientierten Softwareentwicklung*, 5th Workshop on Software Metrics, Berlin, 1995, URL: http://irb.cs.tu-berlin.de/˜zuse/gi/kuhlmann.ps.gz

[15] M. Lorenz, J. Kidd, *Object-Oriented Software Metrics*, Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[16] S. Whitmire, *Object Oriented Design Measurement*, John Wiley & Sons Inc., 1997

[17] M. Xenos, D. Stavrinoudis, K. Zikouli, D. Christodoulakis, *Object-Oriented Metrics - A Survey*, Proceedings of the FESMA 2000, Federation of European Software Measurement Associations, Madrid, Spain, 2000, URL: http://edu.eap.gr/pli/pli10/info/xenos/Personal_page_files/C14 - Object Oriented Metrics - a Survey.pdf

[18] H. Zuse, *Properties of Object-Oriented Software Measures*, Proc. 7th Annual Oregon Workshop on Software Metrics, 1995

[19] H. Zuse, *A Framework of Software Measurement*, de Gruyter Publ., 1998

Der optimale Software Release Zeitpunkt

Daniel Peintner
Mat. Nr. 98 30 851
daniel.peintner@edu.uni-klu.ac.at

Abstract

Steigender Konkurrenzkampf in der Software Industrie, sich ständig ändernde Bedürfnisse der Kunden, gepaart mit Pflege der Software haben das Timing von neuen Software Releases immer wichtiger für den Erfolg eines Software Unternehmens gemacht. Um solch komplexe Entscheidungen treffen zu können, bedarf es einem gut durchdachten Software Release Management. Hinsichtlich dessen werden wirtschaftliche Entscheidungsfindungen wie technische Durchsetzungsdetails beleuchtet. Getrieben von vielen Studien auf diesem Gebiet, besonders dermaßen vieler Modelle zur Zuverlässigkeitsabschätzung, wird ein Überblick geboten und die verschiedenen Gesichtspunkte bearbeitet.

1 Einleitung

Computer, und die nötige Software zur Steuerung, sind allgegenwärtig geworden in der unseren, modernen Gesellschaft.

Die steigende Nachfrage treibt einen Wettkampf an zwischen immer *fähigeren* Produkten und Lösungen. Um dabei die Zuverlässigkeit nicht aus dem Blickwinkel zu verlieren geht das Augenmerk wieder mehr in Richtung Modellen zur Abschätzung von Fehlern. Besonders wichtig ist dies für *life-critical* Software.

Die Arbeit ist wie folgt gegliedert. Abschnitt 2 gibt einen motivativen Einstieg. Es folgt ein wirtschaftlich forcierter Bereich über das Software Release Management gefolgt von einem Überblick über Technische Modelle, deren Annahmen und Umsetzungen. Abschließend werden einige Schlussfolgerungen gezogen die die Arbeit abrunden sollen.

2 Motivation

Die Entwicklung von zuverlässiger Software ist eines der größten Probleme der Software Industrie. Termindruck, limitierte Ressourcen und unrealistische Anforderungen können die Software Zuverlässigkeit negativ beeinflussen [1].

Nach der Auslieferung des Produktes kann man die Zufriedenheit der Kunden von deren Reaktion ablesen – Problemberichte, Komplimente und Beschwerden sind Indikatoren der Selbigen. Doch ist es zu diesem Zeitpunkt schon zu spät. Software Hersteller benötigen diese Informationen früher, bevor sie den Kunden betreffen. Software Reliability Growth Models können helfen diese Informationen zu beschaffen.

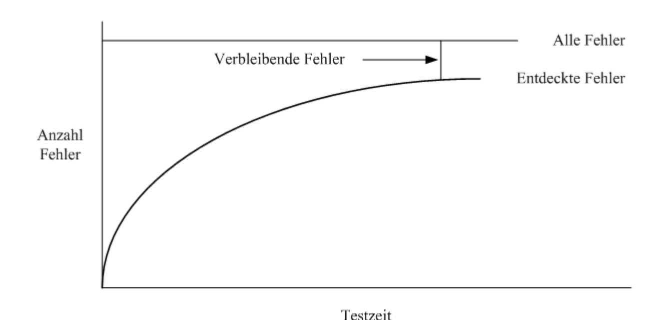


Abbildung 1 – Verbleibende Fehler

3 Das Software Release Management

Software ist wichtiger geworden für den Erfolg eines Unternehmens [3]. Die anwachsende Nachfrage an Software hat einen Wettstreit zwischen den Händlern ausbrechen lassen.

Der Mangel an disziplinierten und vorhersehbaren Software Praktiken ist einer der Hauptgründe für die Überschreitung der Kosten in Software Projekten [3]. In naher Vergangenheit hat die Industrie, auf Grund wachsender Konkurrenz, damit angefangen die Kundenzufriedenheit zu *messen* und nutzt deren Input für Produkt-Entwicklungen und Erweiterungen.

Die Zuverlässigkeit von Software Systemen muss analysiert werden. Das zukünftige Fehlerverhalten ist vorhersehbar, indem man das vergangene Fehlerverhalten studiert und modelliert [2].

Analysten im Software Business betonen die Tragweite der zeitlichen Einführung am Markt. Die immer kürzer werden Release Zeiträume und die wechselnden Benutzer Bedürfnisse verstärken dies noch.

3.1 Wirtschaftliche Aspekte

Software Produkte tendieren dazu jedes Jahr um mindestens zehn Prozent zu wachsen [3]. Die Alterung einer Software ist dabei besonders von Bedeutung. Der optimale ökonomische Kompromiss zwischen Wartung der existierenden Software und dem optimalen Zeitpunkt für die Neuentwicklung ist dabei besonders schwer zu eruieren.

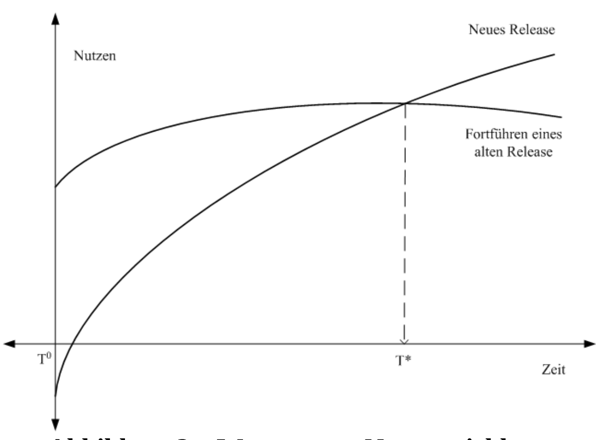


Abbildung 2 – Wartung vs. Neuentwicklung

Der Grund der wachsenden Wichtigkeit der zeitlich optimalen Freigabe rührt von dem Fakt, dass Software zum richtigen Zeit am Markt mehr Marktanteil und größeren Gewinn verspricht. Darüber hinaus bedeutet die Berücksichtigung von Anforderungen von existierenden Kunden in einem neuen Release eine Sicherung des Kundenstammes und ein Vorteil gegenüber der Konkurrenz. Adäquate Zeitintervalle zwischen den einzelnen Produkt-Freigaben sichert weiters Kundenzufriedenheit in Bezug auf berichtete Probleme und Variabilität.

Werden vorgeschlagene Erweiterungen verspätet am Markt positioniert, kann dies bedeuteten, dass ein Teil der Kunden bereits zu Konkurrenzprodukten abgewandert ist. Andererseits kann es aber auch passieren, dass wenn ein Produkt zu früh am Markt erscheint, die Kosten den zusätzlichen Änderungen gegenüber der Vorversion nicht entsprechen.

Eine optimale Freigabe Politik kann man unter vielen Gesichtspunkten betrachten. Beruhend auf Kostenkriterien, Zuverlässigkeitskriterien, wie auch ein Mix der Zuvorgenannten unter Berücksichtigung der Effizienz.

3.2 Technische Aspekte

Software Reliability Growth Models sind deshalb entwickelt worden und den optimalen Zeitpunkt bestimmen zu können, um mit dem Testen aufzuhören.

4 Software Reliability Growth Models

4.1 Begriffsbestimmung [4]

Man spricht von einem *Software Failure*, wenn das Verhalten der Software von dem seiner Spezifikation abweicht. Es handelt sich dabei um das Resultat eines *Software Fault*.

Man kann zwei Aktivitäten in Bezug bringen mit der Zuverlässigkeits-Analyse, *Estimation* (Abschätzung) und *Prediction* (Prognose).

Das Erstere funktioniert rückschauend und wird ausgeführt um von einem Punkt in der Vergangenheit die erreichte Zuverlässigkeit in der Jetztzeit zu erfahren.

Die Prognose andererseits parametrisiert Reliability Models um zukünftige Zuverlässigkeiten vorherzusehen.

4.2 Modelle

Im Allgemeinen kann man Software Reliability Models klassifizieren in *Black Box* und *White Box* Modelle.

Der Unterschied ist einfach der, dass White Box Models die Struktur der Software betrachten um eine Aussage über deren Zuverlässigkeit zu treffen, Black Box Models tun dies hingegen nicht.

4.2.1 Black Box Software Reliability Models. Alle SRGMs sind von diesem Typ, solange sie nur die Fehlerdaten berücksichtigen oder Metriken die gesammelt wurden, wenn keine Testdaten vorhanden sind. Da sie die Software als eine Einheit sehen, bezeichnet man sie als Black Box.

Einige wichtige Begriffe auf diesem Gebiet seien kurz aufgelistet. Die einzelnen Modelle zu präsentieren würde den Rahmen dieser Arbeit sprengen.

Tabelle 1 - Terminologie

Begriff	***Erläuterung***
$M(t)$	Die gesamte Anzahl von Fehlern zum Zeitpunkt t
$\mu(t)$	Durchschnittswert für SRGM. Repräsentiert die erwarteten Fehler zum Zeitpunkt t
$\lambda(t)$	Fehler Intensität, abgeleitet von der Durchschnittswert-Funktion
$Z(\Delta t / t_{i-1})$	Risiko Rate der Software. Die Wahrscheinlichkeitsdichte des i-ten Fehlers
N	Initiale Anzahl von Fehlern vor dem Testen

4.2.2 White Box Software Reliability Models. Man betrachtet die interne Struktur um Abschätzungen treffen zu können. Die Behauptung existiert, dass Black Box Testing nicht adäquate ist im Bereich Component-Based Software. Befürworter von White Box Models behaupten sogar um einiges realistischere Schätzungen damit durchführen zu können.

Der Weg wie solche Modelle benutzt werden lässt sich in folgende drei Klassen einteilen. Pfadbasierte, Zustandsbasierte und Additive Modelle.

4.3 Annahmen vs. Realität [1]

In der Entwicklung von Software Reliability Growth Models werden gewisse Annahmen getroffen. Der Grund dafür liegt auf der Hand. Einerseits versucht man die Realität mit Modellen anzunähern, andererseits müssen diese jedoch auch noch praktisch durchführbar sein, um eingesetzt zu werden.

Im Folgenden seien einige Annahmen beispielhaft kurz angeführt und erläutert. Dass nicht jede Annahme für jedes Modell gilt und es modellspezifisch noch viele weitere Einschränkungen gibt liegt auf der Hand.

Fehler werden sofort berichtigt nachdem sie entdeckt wurden. Dies hieße man müsste das Testen so lange auf Eis legen, solange der Fehler nicht bereinigt wurde. Normalerweise wird aber weiter getestet. In der Realität kommt es vor dass Fehler mitunter nicht sofort ausgemerzt werden können. Damit würde das Modell von einer verzerrten Fehlerfindungseffizienz ausgehen und die Testzeit variieren.

Fehler werden anstandslos behoben. Damit meint man dass sich keine zusätzlichen Fehler bei der Korrektur einschleichen. Studien belegen Gegensätzliches. Man geht davon aus dass zwischen jedem 5ten bis 20ten Fehler der ausgebessert wird ein neuer Fehler produziert wird [1]. Dieses Problem ist bekannt wird aber von den meisten Modellen ignoriert.

Es kommt zu keinem zusätzlichen Code während der Test Periode. Damit meint man einerseits keine zusätzlichen Features und andererseits keine Fehlerbereinigung. Dass der letzte Punkt zum Widerspruch zum ersten steht sei ohne Zweifel. Des Weiteren ist es in vielen Fällen schlicht unmöglich zusätzliche Funktionalitäten zu unterbinden.

Jede Testeinheit hat dieselbe Wahrscheinlichkeit Fehler ans Licht zu bringen. Es kommt aber vor, dass einige wenige Einheiten Bereiche vom Code prüfen, die selten benutzt werden. Dabei ist die Wahrscheinlichkeit höher Fehler zu finden als normal.

Es gibt noch viele weitere Einschränkungen die beachtet werden müssen um ein Modell *einwandfrei* anwenden zu können. Dies ist jedoch von Modell zu Modell recht verschieden und muss problemabhängig betrachtet werden.

4.3.1. Effekte bei Verletzung der Annahmen. Es können folgende Effekte bei Nichteinhaltung verursacht werden.

(1) Der Parameter *Alle Fehler* steigt anstatt konstant zu bleiben. Software Reliability Growth Models gehen davon aus dass die Fehlererkennungsrate abnimmt, nachdem ja jeder Fehler sofort behoben wird und die Anzahl der verbleibenden Fehler immer geringer wird. Da aber zusätzlich auch Fehler hinzukommen, kann die Fehlererkennungsrate geringer sinken als angenommen. Eine andere Möglichkeit besteht darin dass weniger Fehler gefunden werden als prognostiziert. Der Grund dafür kann darin liegen, dass der Fehler erst nach dem Test eingebaut wurde.

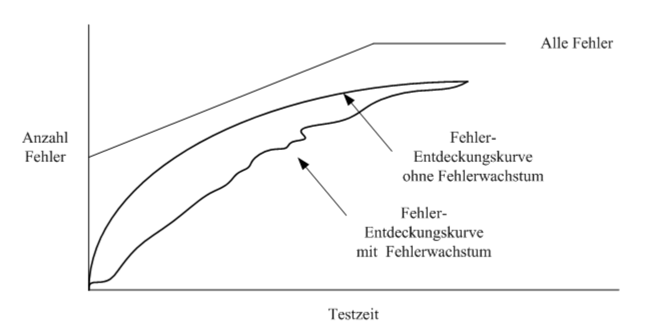


Abbildung 3 – Fehler-Entdeckungskurve

(2) Fehler korrelieren nicht genau mit der Testzeit, weil es zu manchen Zeiten wahrscheinlicher ist Fehler zu finden als zu anderen Zeiten. Eine mathematische Grundvoraussetzung ist aber, dass die Fehlerfindungs Effizienz für jede Testeinheit die Selbe ist. Bei Verletzung dieser Annahme ist eine akkurate Prognose über verbleibende Fehler nicht möglich. Dies würde die Kurve in Abbildung 3 mehr oder weniger konkav formen.

(3) Man kann keine Vergleiche ziehen zwischen unterschiedlichen Releases, da die Erfolgsquote derselben Testfolge für zukünftige Releases weniger wahrscheinlich ist.

4.3.1. Kompensierung bei Verstoß gegenüber Modell Annahmen. Es gibt zumindest drei Möglichkeiten darauf zu reagieren. Verstöße zu ignorieren, Daten zu modifizieren oder neue Modelle abzuleiten.

Der erste Ansatz wird oft gewählt, da man eine gewisse Ungenauigkeit gerne in Kauf nimmt im Austausch für Einfachheit. Parameter Abschätzungen helfen dabei dessen Ausmaße zu bestimmen.

Wenn Annahmen sehr starke Ungenauigkeiten verursachen, besteht die Möglichkeit der Adaptierung von Eingabe Werten. Beispielsweise können die Anzahl der gesamten Fehler wie Teststunden modifiziert werden.

Die bei weitem komplexeste Möglichkeit ist ein für sich geeigneteres Modell zu entwickeln, das der Testumgebung mehr entspricht.

Der Vorteil dabei, besonders bei Zustandsbasierten Modellen, ist dass das Framework zur Zuverlässigkeits- Prognose auch zur Performance Analyse benutzt werden kann.

4.4 Praktische Umsetzung

In den meisten Fällen finden Modelle Einsatz, die auf die jeweilige (Test-)Umgebung adaptiert worden sind. Grundsätzliche gibt es wenige unterschiedliche Ansätze. Beispielsweise für Ansätze die auf die Poisson Verteilung aufbauen, Non-Homogeneous Poisson Process (NHPP) wie auch Enhanced NHPP (ENHPP) Modelle.

Für Fälle wo keine Fehler Daten zur Verfügung stehen, finden statistische Techniken wie Maximum Likelihood Estimation (MLE) großen Anklang.

5 Schlussfolgerung

Wie der Arbeit zu entnehmen ist, stellt die Zuverlässigkeits-Analyse eine entscheidende Rolle im Software Qualitätsprozess dar. Um den optimalen Release Zeitpunkt zu finden, reicht dies aber bei Weitem nicht aus. In den meisten Fällen werden solche Entscheidungen durch firmenpolitische Faktoren mit beeinflusst. Unterstützend einwirken können darauf Software Reliability Modelle die dazu beitragen abzuschätzen wie viele Fehler sich noch in einem Produkt befinden. Man muss den Tradeoff wagen auch in gewissen Situationen lieber erstmal ein nicht ganz fertiges Produkt auf den Markt zu bringen bevor die Konkurrenz sich in seinem Kundenkreis ausbreitet.

Ich würde deshalb meinen, dass der Einsatz von Software Reliability Growth Models sehr wichtig ist. Man wird in vielen Fällen Anpassungen vornehmen müssen, um die eigene Systemlandschaft nachzubilden. Einen groben Aufschluss kann man daraus aber auf jeden Fall ziehen.

6 Referenzen

[1] Alan Wood, "Software Reliability Growth Models: Assumption vs. Reality", *Eighth International Symposium on Software Reliability Engineering (ISSRE '97)*, IEEE, November 1997, pp. 126.

[2] Chin-Yu Huang, Sy-Yen Kuo, Michael R. Lyu, "Optimal Software Release Policy Based on Cost and Reliability with Testing Efficiency", *Twenty-Third Annual International Computer Software and Applications Conference*, IEEE, October 1999, pp. 468.

[3] Mayuram S. Krishnan, "Software Release Management: A Business Perspective", *CASCON '94*, IEEE, 1994.

[4] Ganesh J Pai, "A Survey of Software Reliability Models", Department of ECE, University of Virginia, December 2002

[5] Rong-Huei Hou, Sy-Yen Kuo, Yi-Ping Chang, "Optimal Release Times for Software Systems with Scheduled Delivery Time Based on the HGDM", *IEEE Computer Society*, IEEE, February 1997, pp. 216-221.

[6] Chin-Yu Huang, Jung-Hua Lo, Sy-Yen Kuo, Michael R. Lyu, "Software Reliability Modeling and Cost Estimation Incorporating Testing-Effort and Efficiency", *10th International Symposium on Software Reliability Engineering*, IEEE, November 1999, pp. 62.

[7] Chin-Yu Huang, Sy-Yen Kuo, Ing-Yi Chen, "Analysis of a Software Reliability Growth Model with Logistic Testing-Effort Function", *Eighth International Symposium on Software Reliability Engineering (ISSRE '97)*, IEEE, November 1997, pp. 378.

[8] Chin-Yu Huang, Sy-Yen Kuo, Ing-Yi Chen, "Pragmatic Study of Parametric Decomposition Models for Estimating Software Reliability Growth", *The Ninth International Symposium on Software Reliability Engineering*, IEEE, November 1998, pp. 11.

Zeitmanagement im individuellen Softwareentwicklungsprozess – unter spezieller Berücksichtigung schulischer Aspekte

Katharina Fritz
0060308
kfritz@edu.uni-klu.ac.at

Marina Glatz
0060330
mglatz@edu.uni-klu.ac.at

Abstract

In dieser Arbeit nähern wir uns dem Thema „Zeitmanagement im individuellen Softwareentwicklungsprozess“ sowohl von theoretischer als auch praktischer Seite und erläutern, wie Zeitmanagement in der Schule und im Beruf funktionieren könnte.

Als erstes ist es natürlich nötig, sich mit den Begriffen „individueller Softwareentwicklungsprozess“ und „Zeitmanagement“ auseinanderzusetzen. Ziel hierbei ist es, zu thematisieren, warum durch eine genaue Schätzung des Entwicklungsaufwands der einzelnen Phasen des Softwareentwicklungsprozesses und der daraus resultierenden Zeitplanung qualitativ hochwertige Software entstehen kann.

In weiterer Folge konzentrieren wir uns auf das Zeitmanagement, wobei Methoden vorgestellt werden, wie man die benötigte Zeit für einzelne Prozesse in Projekten eruieren beziehungsweise schätzen kann.

Die Tauglichkeit dieser Techniken wird anschließend sowohl für den Bereich der professionellen Softwareentwicklung als auch den schulischen Bereich analysiert.

1. Einleitung

Nur ein sorgfältig geplantes Softwareprojekt wird gute Ergebnisse erzielen. Deshalb ist es wichtig, im Rahmen des Studiums und auch schon in der Schule, Techniken des Zeitmanagements kennen zu lernen. Ziel dieser Arbeit ist es, den Leser zu motivieren, Zeitmanagementpraktiken in der Schule und im Beruf (in der Softwareentwicklung) einzusetzen.

2. Der individuelle Softwareentwicklungsprozess

2.1. Definition

Unter dem individuellen Softwareentwicklungsprozess versteht man die Abfolge der Arbeitsschritte, die ein Individuum bei der Entwicklung von Software, sei es nun alleine oder als Mitglied in einem Team in einem Unternehmen, an einer Universität oder an einer Schule in einer gewissen Zeit, in einem gewissen finanziellen Rahmen und mit einer gewissen Qualität erbringen muss.

Die Tätigkeiten eines Entwicklers / eines Entwicklungsteams umfassen hierbei die Planung des Projekts aufgrund der Anforderungen eines Auftraggebers, das Design, die Implementierung, die Übersetzung und den Test der Software. Die einzelnen Schritte werden dokumentiert und bilden die Basis für den anschließenden Post-Mortem-Prozess [1], in dem Rückschlüsse auf die Arbeitsweise durch eine Analyse des realen Zeitaufwandes und der aufgetretenen Fehlerraten gezogen werden können.

2.2. Bedeutung des Individuums im Softwareentwicklungsprozess

Jede Software ist nur so gut wie der schlechteste Mitarbeiter. Besonders bei größeren, kostspieligen Projekten (die im Team bearbeitet werden) sind die Anforderungen an den Einzelnen hoch: Ein Entwickler sollte deshalb in der Lage sein, seine Arbeit zu dokumentieren und zu analysieren – dabei seine Stärken und Schwächen zu erkennen und letztere auszumerzen, indem er seine Arbeitsweise in den betreffenden Situationen zu ändern versucht [4].

Für das Zeitmanagement bedeutet das: Um eine rechtzeitige Fertigstellung der Software zu garantieren, muss jeder Einzelne seine Aufgaben pflichtgerecht erledigen. Dazu muss jeder Einzelne genau über seine Arbeitsweise und seine Leistungsfähigkeiten innerhalb

eines gewissen Zeitrahmens bescheid wissen. Weiters wird von jedem Mitarbeiter erwartet, dass er seine Arbeitszeit effektiv und effizient, aber vor allem realistisch plant. Denn nur so kann ein möglichst realistischer Projektplan erstellt und im Verlaufe des Projektes eingehalten werden.

Verfahren, die in Folge vorgestellt werden, können dabei hilfreich sein.

3. Zeitmanagement

3.1. Zeitmanagement als Basis qualitativ hochwertiger Software

Eine wichtige Rolle in der Softwareentwicklung spielt sicherlich der Zeitfaktor. Es ist offensichtlich, dass durch eine mangelhafte Zeitplanung dem Kunden, aber auch dem entwickelnden Unternehmen, massive Kosten erwachsen, welche das Unternehmen in den finanziellen Ruin treiben können. Aber auch der schlechte Ruf, den ein Unternehmen durch eine nicht zeitgerecht gelieferte Software erhält, kann vernichtende Auswirkungen haben.

Um dem vorzubeugen, empfiehlt es sich, möglichst schon vor einem Vertragsabschluss einen (möglichst konkreten) Zeitplan zu erstellen, der sowohl dem Kunden als auch dem Unternehmen eine gewisse Sicherheit bietet. Denn nur Software, der genügend Zeit gewidmet wird, funktioniert (im Allgemeinen) zuverlässig, da sie optimal geplant, designed, programmiert, getestet und dokumentiert wird.

4. Projektplanung

Für jeden angehenden Informatiker, aber auch Informatiklehrer, stellt sich früher oder später die Aufgabe, das erste Projekt zu planen/leiten. Im Allgemeinen kann man hier leider nicht auf Zeitpläne zurückgreifen, an denen man sich orientieren könnte. Auch die Erfahrung, wie viel Arbeit man in einer gewissen Zeit (fehlerfrei) erledigen kann und wie viel Zeit die einzelnen Aufgaben in Anspruch nehmen, fehlt (meist).

Im industriellen Bereich bieten sich für die Zeitschätzung veröffentlichte Daten unterschiedlicher Softwareunternehmen bezüglich der Produktivität, die z.B. in Lines of Code (LOC) angegeben wird, an. Im persönlichen Softwareentwicklungsbereich erweisen sich diese Daten jedoch als unbrauchbar. Jedoch weiß hier der einzelne über sein Leistungspotential relativ genau Bescheid. Allerdings empfiehlt es sich hier auch, Methoden, die im Folgenden vorgestellt werden, anzuwenden, um eine objektivere Sicht auf die Leistung zu erhalten.

Eine Methode, die man anwenden kann, wenn keine Daten vorliegen, wollen wir in diesem Kapitel, insbesondere für angehende Informatiker, kurz erläutern. Es wird erklärt, wie viel Zeit die einzelnen Phasen des Wasserfallmodells beanspruchen und welche Arbeit in einer gewissen Zeitspanne realistisch erscheint und korrekt ausführbar ist. Allerdings beinhaltet dieses Modell auch einige Mängel, wie etwa den fehlenden Einbezug des Software-Reuse und / oder individueller Erfahrungen.

4.1. Aufwandsabschätzung für die einzelnen Phasen des Wasserfallmodells

Das Wasserfallmodell umfasst sechs Phasen, die jeweils mit der Fertigstellung eines Dokumentes abgeschlossen sind [3].

Über den Anteil an Entwicklungszeit, die jede einzelne Phase benötigt, gibt es jedoch unterschiedliche Meinungen. So schlägt Ricketts beispielsweise eine Aufteilung wie in Abb. 4.1 für neue Projekte, über deren Ablauf das Unternehmen noch keine Erfahrungen aus ähnlichen Projekten hat, vor [3].

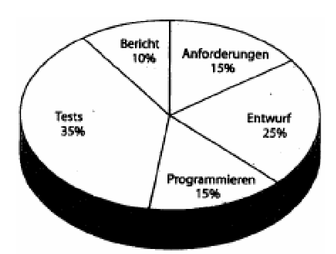


Abb. 4.1. Projektaufwand

Thaller hingegen schätzt den Entwicklungsaufwand wie in Tab. 4.2 beschrieben [4]. Jedoch betonen beide, dass es sich hierbei nur um Richtlinien handelt.

Tab. 4.2. Phasen des Wasserfallmodells

Phase	Dokument	Entwicklungszeit in %
Anforderungsanalyse	Anforderungsdefinition	15
Planung	Gantt Chart (bspw.)	2
Entwurf/Design	Anforderungsspezifikation	20
Implementierung, Kompilierung	Entwurfsvorgaben	36
Test	Implementierungsvorgaben	25
Post Mortem		2

Es fällt auf, dass Thaller auch die Planung in die Entwicklungszeit mit einbezieht – sie also als integralen Bestandteil des Softwareentwicklungsprozesses betrachtet.

4.1.1. Anforderungen: Anforderungen des Kunden sollten unbedingt schriftlich festgehalten werden, denn aufgrund dieser Aufzeichnungen kann man bereits die benötigten Ressourcen und die benötigte Zeit zur Erfüllung des Kundenwunsches berechnen und den Zeitpunkt des Projektbeginns und –endes festlegen. Diese Anforderungen werden in einem Projektplan festgehalten und dienen als Nachweis über erbrachte Leistungen sowohl für den Kunden als auch das Unternehmen [1].

Wenn man den Projektplan erstellt, sollte auf jeden Fall berücksichtigt werden, dass man pro Woche höchstens fünf Arbeitstage und pro Tag höchstens 8 Arbeitsstunden einplant. So bleibt für eventuelle Sonderfälle, ungeplante Aktivitäten und / oder ungenaue Schätzungen genügend Zeit um zu reagieren [3].

4.1.2. Entwurf: Für eine realistische Zeitplanung ist es wichtig, schon vor der Implementierung abzuschätzen, wie viel Code man programmieren kann. Das geschieht in der Entwurfsphase.

4.1.3. Implementierung: Ein Softwareentwickler sollte in der Lage sein, ungefähr 60 Zeilen kommentierten Code pro Tag zu schreiben – vorausgesetzt, ein ausführlicher Entwurf der Software besteht bereits und das Testen und die Fehlersuche sind eigene Aktivitäten.

Das sauber entworfene Programm sollte aus Modulen bestehen. [3]

4.1.4. Test: „Fehlerhafte Software wird oft als unzuverlässig betrachtet und deshalb nicht verwendet." [2] Aus diesem Grunde ist es wichtig, dass man ausreichend Zeit zum Testen der einzelnen Methoden verwendet.

4.1.5. Projektbericht: Wurde während der einzelnen Phasen wie geplant immer wieder dokumentiert, so schreibt man in dieser Phase zirka zehn Seiten Bericht pro Tag, da man nur noch die Berichte zusammenfügen muss [3].

4.2. Entwicklung eines Zeitplans

Sowohl Humphrey [1] als auch Thaller [4] und Ricketts [3] schlagen Zeitpläne in Form von Gantt Charts (Abb. 4.3) vor. Diese Pläne bestehen aus (stündlichen oder wöchentlichen) Meilensteinen, die verdeutlichen sollen, wann eine gewisse Phase des Wasserfallmodells (oder in der Schule eines größeren Projekts) fertig gestellt sein muss. Während bei einem Projekt, das nur von einer Person durchgeführt wird, keine Synchronisation des Planes notwendig ist, ist eine solche bei Teamarbeit von großer Bedeutung.

Zeitpläne dürfen in keinem Projekt fehlen, da sie jeden einzelnen (mehr oder weniger) dazu zwingen, sich an die darin enthaltenen (zeitlichen) Vereinbarungen (gekennzeichnet durch Meilensteine) zu halten. Durch die Meilensteine bietet sich auch die Möglichkeit genau zu prüfen, wo man sich in einem Projekt befindet und bei Verzug, rechtzeitig zu reagieren.

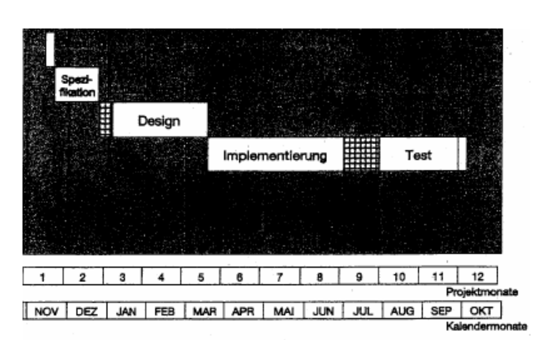


Abb. 4.3. Gantt Chart

5. Individuelles Zeitmanagement

5.1. Methoden der effizienten Zeitplanung

Es ist sehr wahrscheinlich, dass man immer wieder ähnlichen Tätigkeiten, sowohl im Berufs- als auch im Privatleben, Woche für Woche nachgeht. Dennoch sollte man versuchen, möglichst genau aufzuzeichnen, womit man seine Zeit verbringt, denn oft trügt das intuitive Zeitgefühl.

Aus diesen Aufzeichnungen kann man in weiterer Folge einen Zeitplan erstellen und versuchen seine Zeit noch besser einzuteilen. Hierbei ist es vor allem wichtig, dass man den Plan konsequent verfolgt und falls er sich als zu ungenau herausstellt, anpasst.

Im Folgenden werden einige von Humphrey [1] entwickelte Methoden vorgestellt, die dabei helfen können, die Zeit sowohl für Softwareentwickler als auch Schüler, realistisch zu planen.

5.1.1. Engineering Notebook: Um festzustellen, wie man seine Zeit nutzt, ist es hilfreich, die momentanen, unzähligen Aktivitäten zu kategorisieren, damit der Zeitplan übersichtlich auf einige wenige (aber wichtige) Hauptkategorien, die bei Bedarf noch verfeinert werden können, beschränkt bleibt.

Das Engineering Notebook dient vor allem dazu, eine chronologische Ordnung in die Aufzeichnungen zu bringen. Als sinnvoll entpuppt sich daher ein von ihm vorgeschlagenes Inhaltsverzeichnis, da es die Übersichtlichkeit im Notebook erhöht.

Ein Engineering Notebook kann aber auch Notizen zu Gesprächen, Vorlesungen und Designanmerkungen, das Time Recording Logbuch (siehe 5.1.2.) sowie Ver-

weise auf Prüfungen und Vereinbarungen enthalten. Im Falle einer Klage, kann es dem Unternehmen auch als Beweismittel dienen.

Beispiel: Ein Engineering Notebook Eintrag, der die (Haupt-)Kategorie „Hausaufgaben" dokumentiert:

Date	3
9/9	CSI assignment, due 9/16
	Make an engineering notebook
	Reference, page 206, textbook
	Read programming text, chapter 1
9/11	CSI assignment, due 9/20
	Do programming exercises, chapter 1
9/13	CSI assignment due 9/23
	Read programming text chapter 2
	Review the exercises in chapter 2 and prepare for quiz

Abb. 5.1. Engineering Notebook

5.1.2. Time Recording Logbuch: Das Time Recording Logbuch bietet die Möglichkeit, Rückschlüsse auf seine Arbeitsweise zu ziehen, indem man die Zeit, die man für einzelne Tätigkeiten während eines Projektes benötigt, misst.

Ist man in mehrere Projekte gleichzeitig involviert, wird für jedes einzelne ein eigenes Logbuch angelegt. Dasselbe könnte man auch im Projektunterricht an der Schule durchführen.

Die Zeitmessung erfolgt in Minuten. Neben der Zeitnahme für die eigentlichen Aufgaben (wie etwa programmieren) ist es ebenfalls unbedingt nötig, die Dauer „banaler" Unterbrechungen wie Kaffeepausen oder E-Mail-Bearbeitung während einer Aktivität aufzuzeichnen. Es ist ebenfalls wichtig, den Beginn und das Ende, sowie die Nettozeit (also die Zeit ohne Unterbrechungen) der Bearbeitung eines Problems schriftlich festzuhalten.

Date	Start	Stop	Interruption Time	Delta Time	Activity	Comments	C	U
9/9	9:00	9:50		50	Class	Lecture		
	12:40	1:18		38	Prog.	Assignment 1		
	2:45	3:53	10	58	Prog.	Assignment 1		
	6:25	7:45		80	Text	Read text - Ch 1&2	X	2
9/10	11:06	12:19	6+5	62	Prog.	Assignment 1, break, chat	X	1
9/11	9:00	9:50		50	Class	Lecture		
	1:15	2:35	3+8	69	Prog.	Assignment 2, break, phone	X	1
	4:18	5:11	25	28	Text	Text Ch 3, Chat with Mary	X	1
9/12	6:42	9:04	10+6+12	114	Prog.	Assignment 3	X	1
9/13	9:00	9:50		50	Class	Lecture		
	12:38	1:16		38	Text	Text Ch 4		
9/14	9:15	11:59	5+3+22	134	Review	Quiz prep, break, phone, chat		

Abb. 5.2. Time Recording Log

Zum besseren Verständnis des Time Recording Logs, wird der Eintrag am 10. September genauer erläutert:

Neben dem Datum, dem Beginn und dem Ende der Aktivität, werden allfällige Unterbrechungen (hier: 6+5 Minuten) sowie die Art der Unterbrechung angeführt. Die Nettozeit sowie die Art der Aktivität sind ebenfalls aufzulisten. Durch das x und die 1 erfährt man, dass genau ein Kapitel erfolgreich fertig gestellt wurde.

5.1.3. Periodenpläne: Ausgehend vom Time Recording Logbuch können die darin gesammelten Daten zur Erstellung von Periodenplänen genutzt werden. Die Hauptaufgabe von Periodenplänen ist die übersichtliche Darstellung der Zeit, die man innerhalb einer Periode (z.B. einer Woche) für die einzelnen Aktivitäten benötigt.

Ausgehend von Abbildung 5.2 lässt sich die Weekly Activity Summary aus Abbildung 5.3. erstellen. So werden z.B. unter Montag, die „Vorlesungszeit", die Programmierzeit und die Lesedauer (ersichtlich aus dem Time Recording Log), verzeichnet.

Name Student Y   Date 9/16/96

1	Task	Class	Write	Quiz	Read					Total
2	Date		Prog.	Prep.	Text					
3	S 9/8									
4	M	50	96		80					226
5	T		62							62
6	W	50	69		28					147
7	T		114							114
8	F	50			38					88
9	S			134						134
10	Totals	150	341	134	146					771

11 Period Times and Rates   Number of Weeks (prior number +1): 1

Abb. 5.3. Weekly Activity Summary I

Das Vorgehen ist auch für die Folgewochen gleich, allerdings kommen Informationen dazu, die helfen die durchschnittliche, maximale und minimale Arbeitszeit pro Aufgabenfeld zu ermitteln. Diese Werte werden in der Abbildung 5.4 festgehalten. Durch diese Informationen können, für kommende Wochen fällige (ähnliche) Arbeiten, zeitmäßig gut eingeschätzt werden.

12 Previous Week's Times

13	Total	150	341	134	146					771
14	Avg.	150	341	134	146					771
15	Max.	150	341	134	146					771
16	Min.	150	341	134	146					771

17 Current Week's Times

18	Total	300	680	134	370					1484
19	Avg.	150	340	67	185					742
20	Max.	150	341	134	224					771
21	Min.	150	339	134	146					713

Abb. 5.4. Weekly Activity Summary II

5.1.4. Produktpläne: Der Hauptunterschied zwischen Periodenplänen und Produktplänen liegt darin, dass sich Produktpläne auf die Dauer, die ein Produkt bis zur Fertigstellung benötigt, bezieht, und nicht die Arbeiten auflistet, die innerhalb einer Periode gemacht werden.

Ein Produktplan sollte über die vermutete Größe, über die benötigte Zeit und den Verlauf des Projektes Auskunft geben, damit ein wohl definiertes und schließlich zuverlässiges Endprodukt gewährleistet werden kann.

Für größere Produkte empfiehlt es sich die Arbeit in kleinere Jobs aufzuteilen, da solche leichter zu planen sind und folglich solche Pläne auch leichter eingehalten werden können.

Wie wird nun ein Produktplan verwendet, um seine Arbeit sinnvoll zu schätzen? Wenn man weiß, dass in dieser Woche eine ähnliche Aufgabe (z.B.: eine Programmieraufgabe) anfällt wie in einer der Vorwochen, so kann man aufgrund der vorhergegangen Aufzeichnungen über die Art des Jobs und dessen tatsächlicher Dauer, die erwartete Zeit für die jetzt fällige Aufgabe genauer schätzen (als ohne eine solche Aufzeichnung). Am Ende wird die Schätzung an der tatsächlichen Dauer der Aufgabe gemessen, die ebenfalls im Time Recording Log (siehe Abbildung 5.2) notiert wird.

Job #	Date	Process	Estimated		Actual			To Date				
			Time	Units	Time	Units	Rate	Time	Units	Rate	Max	Min
1	9/9	Prog	100	1	158	1	158	158	1	158	158	158
	Description: Write program 1 (minutes per program)											
2	9/9	Text	50	2	80	2	40	80	2	40	40	40
	Description: Read textbook Chapters 1 and 2 (minutes per chapter)											
3	9/11	Prog	158	1	69	1	69	227	2	113.5	158	69
	Description: Write program 2											
4	9/11	Text	40	1	28	1	28	108	3	36	40	28
	Description: Read textbook Chapter 3											
5	9/12	Prog	114	1	114	1	114	341	3	113.7	158	69
	Description: Write program 3											
6	9/13	Text	60	1	118	1	118	226	4	56.5	118	28
	Description: Read textbook Chapter 4											

Abb. 5.5. Produktplan auf Basis des Time Recording Logbuchs

Zum besseren Verständnis der Abbildung 5.5 erfolgt eine kurze Erläuterung der Jobs 2 und 6.

Job 2: Zum Lesen des Textes nahm man sich eine Zeitspanne von 50 Minuten vor. Tatsächlich benötigte man für zwei Kapitel 80 Minuten, das sind 40 Minuten pro Kapitel. Statistisch gesehen, verbrachte man bisher 80 Minuten mit Lesen. Das sind je Kapitel durchschnittlich, maximal und minimal 40 Minuten.

Job 6: Man rechnete mit 60 Minuten zum Lesen eines Kapitels. Tatsächlich benötigte man 118 Minuten. Damit brauchte man bisher für 4 Kapitel 226 Minuten, was eine Durchschnittszeit von 56,5 Minuten ergibt. Die nächste Schätzung für Job 9 basiert auf diesem Mittelwert.

Je mehr Daten bezüglich einer Aufgabenart vorhanden sind, desto genauer wird die Prognose in Zukunft ausfallen.

Kritisieren möchten wir an diesem Ansatz trotzdem, dass es für Ungeübte häufig schwer ist, einzuschätzen, inwieweit sich gewisse Aufgaben ähneln. Viele Anfänger (vor allem Schüler) werden beim Durchlesen der Angabe einer Programmieraufgabe, nicht sofort beurteilen können, ob der Schwierigkeitsgrad in etwa gleich dem der Vorwoche ist.

5.1.5. Zeitbudget-Erstellung: Ausgehend von den bisherigen Aufzeichnungen können wir nun ein Zeitbudget erstellen, welches in Folge mit der tatsächlich benötigten Zeit verglichen wird. Sollten größere Abweichungen auftreten, müssen diese analysiert werden und gegebenenfalls nachfolgende Zeitpläne angepasst oder der Arbeitsstil geändert werden.

Activity Category	Budget Minutes	Actual Minutes
Attend Classes	150	
Write Programs	360	
Read Text	180	
Review for Exams	120	
Other	30	
Total	840	

Abb. 5.6. Zeitbudget

In obigem Beispiel will man auch für eine Prüfung lernen. Dazu benötigt man Extrazeit, die von anderen Aktivitäten weggenommen werden muss.

Es kann aber auch passieren, dass neben geplanten Ereignissen (wie Prüfungen) auch unerwartete Sonderfälle eintreten. Solchen Sonderfällen muss man notfalls auch Freizeit opfern. Sollten jedoch ständig Einschränkungen des Privatlebens auftreten, sollte man auf jeden Fall sein Zeitmanagement überprüfen. Denn so viele Sonderfälle kann es gar nicht geben!

Bei der Erstellung von Zeitbudgets sollte man berücksichtigen, dass es neben fixen Zeiten, wie etwa Vorlesungen oder Besprechungsterminen, auch variable Zeit gibt. In dieser variablen Zeit kann man dann vorgeschriebene Aufgaben, wie etwa Hausübungen oder (4-h) Praktika, beliebig einteilen. Sollten die Arbeiten mehr Zeit erfordern, muss man nötigenfalls auch Teile seiner Freizeit dafür opfern.

Weiters ist zu berücksichtigen, dass Arbeiten mit hoher Priorität so schnell wie möglich bearbeitet werden sollten.

5.2. Bewährung der vorgestellten Methoden in Schule und Beruf

Als angehende Lehrerinnen, die bereits das Schulpraktikum absolviert und dadurch Einblick in den Schulalltag aus der Sicht des Unterrichtenden gewonnen haben, sind wir der Meinung, dass die vorgestellten Methoden von Humphrey durchwegs relevant für Schüler und auch Lehrer sind.

Allerdings ergeben sich auch einige Schwierigkeiten in der Umsetzung, die im Weiteren genauer erläutert werden.

Die Akzeptanz bei den Lehrern, Schülern, aber vor allem auch bei den Eltern ist sicherlich ein Faktor, der darüber entscheidet, inwieweit Zeitmanagement wirklich im „geheimen" Lehrplan Einzug findet.

Eine Weigerung, solche Methoden in den Unterricht aufzunehmen, könnte zum einen ein ungewohnter Zeitaufwand für die Planung sein. Zum anderen besteht auch die Gefahr des „gläsernen" Schülers, der unfreiwillig durch die Kontrolle der persönlichen Aufzeichnungen des Schülers entsteht.

Andererseits bleibt dem Lehrer, wenn er diese Methoden einsetzt, keine Alternative, als die Aufzeichnungen zu kontrollieren, da sonst einige Schüler dazu verleitet würden, gar keine Aufzeichnungen zu machen.

Die Kontrolle des Lehrers führt jedoch wiederum dazu, dass viele Schüler „lehrergerechte" Aufzeichnungen anfertigen, die wenig mit dem tatsächlichen Aufwand des Schülers gemein haben.

Weiters wird es schwierig sein, Schüler aber auch Kollegen dazu zu motivieren, ihre alten Muster der (eher intuitiven) Zeitplanung, aufzugeben.

Trotz des anfänglichen hohen Gesamtaufwandes sollte man die Eltern, Kollegen und Schüler dennoch von der Notwendigkeit der vorgestellten Methoden überzeugen, da heute in vielen Berufsfeldern Teamarbeit und projektorientiertes Arbeiten unter harten zeitlichen Grenzen gefordert ist. Schüler, denen ein fundiertes Wissen bezüglich Zeitmanagement (z.B. durch Projektunterricht) rechtzeitig vermittelt wird, werden in der Lage sein, mit größeren Projekten adäquat umzugehen.

Wenn man bereits beruflich in den Softwareentwicklungsprozess eingebunden ist, werden wahrscheinlich ähnliche Probleme wie in der Schule auftreten. Man braucht nur „Lehrer" durch „Vorgesetzten" oder „Projektleiter" und „Schüler" durch „Mitarbeiter" zu ersetzen…

Sofern der Manager davon überzeugt ist, dass qualitativ hohe Software der individuellen Zeitplanung jedes Mitarbeiters, bedarf, wird er jedoch Methoden des Zeitmanagements einsetzen.

Hierbei werden die (doch sehr allgemeinen) Methoden von Humphrey wohl vermehrt auf die Bedürfnisse der Firmen angepasst werden müssen. Auch rigide vorgeschriebene Layouts und Formatierungen sowie die Verwendung bestimmter Utensilien werden mit wachsender Erfahrung wohl aufgegeben werden.

Eine weitere Schwachstelle Humphreys, die vor allem bei größeren Projekten zum Tragen kommt, ist die völlige Vernachlässigung der Komplexität von Aufgaben. Hierfür bietet er nämlich keine Methoden, wie man die Problemgröße eines Projektes adäquat ermitteln kann. Es ist jedoch für viele Menschen nicht auf den ersten Blick ersichtlich, wie komplex ein Projekt ist und wie viel Zeit wirklich für die Ausarbeitung benötigt wird.

6. Conclusio

Zeitplanung ist sowohl im Beruf als auch in der Schule notwendig, um qualitativ hochwertige Produkte zu fertigen. Vor allem in der Softwareentwicklung ist eine sorgfältige Planung ausschlaggebend für den Erfolg der Software. Erprobte Methoden wie die von Humphrey aber auch Thaller und Ricketts können, sofern man sich bei der Entwicklung von Software auch an das Wasserfallmodell oder andere ausgereifte Modelle hält, eine gewisse Sicherheit bieten, dass Projekte fristgerecht und großteils fehlerfrei fertig gestellt werden können.

Das Wasserfallmodell und die dafür entwickelten „Entwicklungszeitprozentangaben" können vor allem unerfahrenen Projektleitern als Anhaltspunkt dienen.

Obwohl wir Zeitmanagement als sinnvoll erachten, werden sich die Methoden von Humphrey nur schwer innerhalb des Informatikunterrichts unterbringen lassen, da der Lehrplan nur wenige Stunden für den Informatikunterricht vorsieht (Lehrplan Informatik unter http://www.bmbwk.gv.at/medienpool/7037/Informatik_Oberstufe.pdf).

Literatur

[1] Humphrey, Watts S.: „Introduction to the Personal Software Process"; Addison-Wesley Publishing Company (2002)
[2] Humphrey, Watts S.: "Managing the Software Process"; Addison-Wesley Publishing Company (1990)
[3] Ricketts, Ian W.: „Software-Projektmanagement kompakt"; Springer (1998)
[4] Thaller, Georg Erwin: "Softwareentwicklung im Team", Galileo Press, Bonn (2002)

Testen komponentenbasierter Software

Thomas FRANK
Matr.-Nr.: 0060762
thomas.frank@edu.uni-klu.ac.at

Abstract

In die komponentenbasierte Software-Entwicklung werden hohe Erwartungen gesteckt. Signifikant geringere Entwicklungskosten und kürzere Entwicklungszeiten werden erhofft. Schließlich sollen die wiederverwendeten Komponenten zu einer höheren Software-Qualität beitragen. Dem Testen kommt hier eine besondere Bedeutung zu. Schwierigkeiten wie das Fehlen des Sourcecodes, der Spezifikationen und/oder der detaillierten Requirements erfordern neue Modelle, die teilweise auf klassische Test-Verfahren aufbauen, teilweise aber an das Problem auf eine neue Art und Weise herangehen.

In der vorliegenden Arbeit werden einige Ansätze besprochen wie das Testen komponentenbasierter Software durchgeführt bzw. erleichtert werden kann. Eingegangen wird dabei auf eine spezielle Art des Repository-Designs, das Regressionstesten erleichtern soll, auf ein Maturity-Model für den komponentenbasierten Test-Prozess sowie auf einen Vorschlag zur Automatisierung des Testens. Weiters werden Modelle vorgestellt, die zur Durchführung der Tests nicht unbedingt auf der Kenntnis des Sourcecodes beruhen.

1. Einleitung

Komponentenbasierte Softwareentwicklung erleichtert die Software-Wiederverwendung und fördert somit Produktivität und Qualität. Nicht nur die Qualität der Software-Komponenten, sondern besonders auch die Effektivität der Testprozesse tragen zur Qualität des Gesamtsystems bei. Gerade auf das Testen im Zusammenhang mit komponentenbasierter Software wurde in letzter Zeit in der Literatur eingegangen [1][2]. Einige Ansätze werden in dieser Arbeit behandelt.

In Kapitel 2 wird ein kurzer Überblick über die komponentenbasierte Softwareentwicklung gegeben. In Kapitel 3 wird eine Abgrenzung bestimmter Begriffe vorgenommen, die die Bereiche „Komponentenbasierte Softwareentwicklung" sowie „Testen" umfassen. Kapitel 4 ist der Hauptteil der Arbeit und stellt die Ansätze des Testens komponentenbasierter Software vor. Einerseits geht es dabei um Modelle bei der die Komponenten als Software-Module behandelt werden (Kapitel 4.1 bis 4.3). Andererseits wird die Komponente als binärer Software-Baustein gesehen. In Kapitel 4.4 werden Lösungsansätze für das Problem des Testens bei fehlendem Sourcecode, das bei Software-Binär-Bausteinen gegeben ist, vorgestellt.

2. Komponentenbasierte Software

Moderne Softwaresysteme werden immer umfangreicher und komplexer, was zu hohen Entwicklungskosten, geringer Produktivität und Problemen mit der Software-Qualität führt. Ein vielversprechender Lösungsansatz ist dafür die komponentenbasierte Softwareentwicklung (Component-Based Softwareengineering – CBSE).

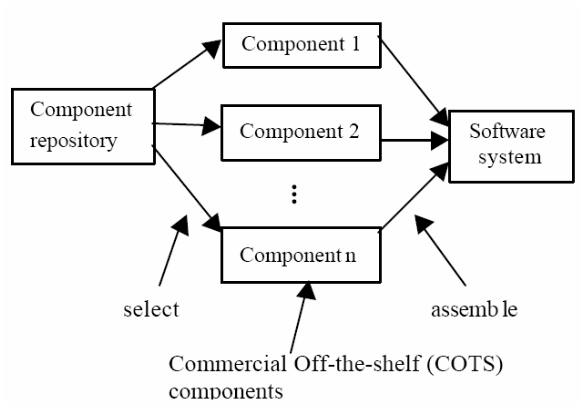


Abbildung 1. Komponentenbasierte Software-Entwicklung [3]

Die Idee ist dabei passende Off-The-Shelf-Komponenten auszuwählen und diese anhand einer klaren Software-Architektur zu einem Software-System zusammenzusetzen (siehe Abbildung 1). Diese Komponenten können von unterschiedlichen Entwicklern kommen, unter Verwendung verschiedenster Pro-

grammiersprachen implementiert sein, sie können auf unterschiedlichen Plattformen und als verteiltes System ausgeführt werden [3]. Diese Heterogenität bringt Probleme mit sich, wenn komponentenbasierte Software mit traditionellen Methoden getestet werden soll.

Wenn der Sourcecode einer Komponenten nicht verfügbar ist, kann ein White-Box-Test normalerweise nicht angewandt werden. Falls der Sourcecode doch vorliegt, können Probleme dadurch auftreten, dass die Komponenten in verschieden Programmiersprachen geschrieben wurden. Ein weiteres Problem kann die Tatsache darstellen, dass durch die Möglichkeit des einfachen Plug-and-Play Dritte ihre Komponenten immer wieder updaten, was jedes Mal Testaufwand erfordert. Testfälle werden oft aus Spezifikationen abgeleitet, welche neben dem Sourcecode ebenfalls nicht immer vorhanden sind.

3. Begriffsbestimmungen

Komponente. Eine Komponente hat drei Hauptmerkmale: 1) eine Komponente ist ein unabhängiger, ersetzbarer Teil eines Systems, der eine eindeutige Funktion erfüllt; 2) eine Komponente arbeitet im Kontext einer wohldefinierten Architektur; 3) eine Komponente kommuniziert mit anderen Komponenten über ihre Schnittstelle [3]. Eine Komponente ist grundsätzlich ein binärer Software-Baustein, man spricht hier meist von Commercial-Off-The-Shelf (COTS) Komponenten. Oftmals werden Komponenten aber auch entwickelt um eine stärkere Modularisierung und damit bessere Wartbarkeit von Anwendungen zu erreichen.

Test-Treiber, Test-Stub, Test-Case, Test-Suite. Test-Treiber stellen einen Testrahmen zur Verfügung, der den interaktiven Aufruf der zu testenden Dienste einer Systemkomponente ermöglicht. Test-Stubs sind Platzhalter um noch nicht implementierte Systemkomponenten zu simulieren. Sie werden bei Integrationstests benötigt. Ein Test-Case ist ein Satz von Testdaten, der die vollständige Ausführung eines Pfades des zu testenden Programms initiert. Eine Test-Suite ist eine Menge solcher Test-Fälle [6].

Unit-Test, Integrations-Test, System-Test. Beim Unit-Test wird die Korrektheit der individuellen Komponenten getestet. Für den Integrations-Test werden mehrere Komponenten zu einem Subsystem zusammengebaut und als Einheit getestet. Beim System-Test wird wiederum aus mehreren Subsystemen das System zusammengestellt und als Gesamteinheit getestet [4].

Überdeckungsgrad, Regressionstest. Der Überdeckungsgrad ist ein Maß für den Grad der Vollständigkeit eines Tests bezogen auf ein bestimmtes Testverfahren. Beim Regressionstest handelt es sich um die Wiederholung der bereits durchgeführten Tests nach einer Änderung im Programm [6].

White-Box-Test, Black-Box-Test. Beim White-Box-Test werden die Testfällen aus dem Sourcecode erzeugt. Die innere Struktur des Programms muss daher bekannt sein. Black-Box-Testen ist ein dynamisches Testverfahren, bei dem die Testfälle aus der Programmspezifikation abgeleitet werden. Die Programmstruktur wird dabei nicht beachtet [6].

4. Modelle/Ansätze des Testens komponentenbasierter Software

Ein Reihe möglicher Modelle bzw. Ansätze des Testens komponentenbasierter Software wird in diesem Abschnitt vorgestellt. Es handelt sich dabei nicht um Alternativen, sondern um mögliche Lösungen einiger der vorher beschriebenen Probleme, die an verschiedenen Stellen ansetzen.

4.1 Repository Design

Weyuker [4] schlägt eine spezielle Art des Repository-Designs vor. Softwarekomponenten müssen so abgelegt werden, dass ein leichtes Auffinden und Wiederverwenden möglich ist. Zusätzlich muss jeder Komponente eine Person oder ein Team zugeordnet werden, die/das für die Wartung verantwortlich ist. Dass sich die Entwickler streng an Interface-Standards halten, ist dabei wesentlich.

Für jede Komponente müssen mehrere Artefakte gespeichert und ständig aktuell gehalten werden:

- Die Spezifikation inklusive Referenzen zwischen Requirements und deren Implementierung. Wenn im Code bzw. der Spezifikation Änderungen gemacht werden, muss diese Änderung im entsprechenden übereinstimmenden Dokument nachgezogen werden.
- Die Test-Suite inklusive Querverbindungen zwischen den Test-Fällen und den Code-Teilen, die damit getestet werden sollen. Neue Funktionalitäten führen zu neuen Test-Fällen in der Test-Suite. Dies erleichtert die Auswahl der Test-Fälle eines Regressionstests.
- Pointer zwischen der Spezifikation und den Teilen der Test-Suite, die diese Funktionalitäten validieren. Dies zeigt in welchem Ausmaß bestimmte Teile der Spezifikation in der Test-Suite berücksichtigt werden.

Auf alle Fälle sollte eine Test-Suite möglichen Input und erwarteten Output enthalten. Dies erleichtert den Regressionstest nach Änderung der Komponenten.

4.2 Maturity-Model für den Komponenten-Test-Prozess

Um die Effektivität der Testprozesse messen zu können, wird von Gao [2] ein fünfstufiges Maturity-Model vorgeschlagen. Level 0 wird als Ad-Hoc-Komponententest beschrieben. Ad-Hoc-Tests sind sehr ineffizient und durch die geringe Wiederverwendbarkeit der Testinformationen, Testtreiber und -stubs auch sehr teuer. Die Qualität der Tests kann nicht kontrolliert und gesteuert werden. Im Level 1, dem Standard-Komponententest, werden Standards definiert, die die Informationsverarbeitung im Testprozess (Testpläne, -berichte, ...), Managementprozesse (Configuration-Management, Qualitätkontrolle, ...) und Testkriterien beschreiben. Level 2 entspricht dem geführten Komponententest. Aktivitäten im Bereich der Messung bzw. Überwachung der Kosten, der Qualität und verschiedener Testmetriken kommen hier hinzu. Level 3 – das zertifizierte Komponententesten – ist dann erreicht, wenn zertifizierte Standards definiert und implementiert wurden. Dazu gehören zertifizierte Testpläne, Testplattformen, -umgebungen, -metriken etc. Und schließlich ist Level 4, das systematische Komponententesten dann erreicht, wenn systematische Methoden und Mechanismen definiert und implementiert sind, die schließlich zu einer Automatisierung der Testprozesse führen [2].

4.3 Test-Automatisierung

Durch die Automatisierung von Prozessen verspricht man sich einiges an Verbesserungen und Einsparungen, so auch beim Testen in der komponentenbasierten Softwareentwicklung. Wie dies möglich wäre hat Jerry Gao [2] untersucht. Die systematische Verwaltung der Test-Informationen ist der grundlegende Schritt in Richtung Testautomatisierung. Standards müssen für alle Bereiche des Testens festgelegt werden. Ebenso muss auf Portabilität und Kompatibilität geachtet werden, um mit verschiedenen Plattformen und Sprachen arbeiten zu können. Wichtig ist hier die Frage wie wiederverwendbare und konfigurierbare Testtreiber und Teststubs für eine Komponente generiert werden können. Testtreiber müssen skriptbasierte Programme sein, die auf Basis von Funktionen oder Szenarien arbeiten.

Zur automatischen Durchführung der Tests ist eine Umgebung nötig, die drei wichtige Bereiche umfasst:

- Ein Komponenten-Testcontroller der aus einer Testsuite die Testskripts generiert, die Ergebnis überprüft und Berichte erstellt.
- Ein Komponenten-Test-Stub-Manager der aus einem Test-Stub-Controller, einer Schnittstelle zu einem Test-Stub-Repository und einem Generator für Test-Stubs besteht.
- Eine Testumgebung die die Interaktion der Komponenten mit dem Komponenten-Testcontroller und dem Test-Stub-Manager unterstützt.

Plattformunabhängige Technologien, wie Java, werden vorgeschlagen um eine solche generische Black-Box-Test-Umgebung zu bauen. Angaben inwieweit solche Versuche der Automatisierung bereits realisiert wurden macht Gao nicht [2].

4.4 Testen ohne Sourcecode

Eine der wesentlichen Einschränkungen beim Testen komponentenbasierter Software ist oft das Nichtvorhandensein des Quellcodes. Für User von COTS-Komponenten besteht eine verzwickte Lage. Auf der einen Seite stehen Einsparungen von Kosten bzw. Verkürzung der Entwicklungszeit, andererseits benötigen sie zur Sicherstellung der Qualität der Komponenten aber Informationen, die die Hersteller der Komponenten nicht weitergeben können, um auf dem Markt nicht geistiges Eigentum und somit vielleicht entscheidende Wettbewerbsvorteile an die Konkurrenz zu verlieren. Besonders deutlich tritt dies bei sicherheitskritischen System hervor. Die Frage ist hier auch wie Kunden trotz des Schutzes geistigen Eigentums von einem erreichten Überdeckungsgrad überzeugt werden können.

Black-Box Testen. Voas [7] meint der White-Box-Test ist aufgrund des Mangels an Informationen über das Systems auszuschließen. Er schlägt drei Techniken vor:

- *Reines Black-Box-Testen*. Anhand der am häufigsten auftretenden Szenarien wird das korrekte Verhalten der Komponente geprüft. Dieser Ansatz hat allerdings einige Nachteile. Eine sehr große Zahl von Tests muss durchgeführt werden um die Verlässlichkeit der Komponenten abschätzen zu können. Falls keine automatisierte Bewertung des Testoutput möglich ist, kann dieser Ansatz sehr teuer werden.
- *System-Level-Fault-Injection*. Dieser Ansatz prüft die Auswirkungen abnormalen Verhaltens der Komponenten auf das Gesamtsystem. Dieses Verhalten wird durch Veränderung der Eingaben/Ausgabe-Beziehungen erreicht. Dieser Ansatz untersucht die Robustheit des Gesamtsystems. Die

Kosten die durch Erhöhung der Robustheit entstehen müssen im Verhältnis zu den Kosten für die Komponenten stehen.

- *Operational-System-Testing*. Dabei handelt es sich um einen erweiterten Systemtest, wodurch die Reliability eines Gesamtsystems mit eingebauten COTS-Komponenten abgeschätzt werden soll. Mit einer Vielzahl an zufälligen Samples wird das System getestet. Ein Nachteil ist eben diese große Sample-Menge, die nötig ist, um gute Schätzungen machen zu können. Dies kann teuer und zeitaufwendig sein.

White-Box Testen. Devanbu und Stubblebine [5] entwickelten einen Ansatz für das White-Box-Testen mit hohem Überdeckungsgrad mit Rücksichtnahme auf den Schutz geistigen Eigentums. Sie schlagen vor den Test von einem Dritten durchführen zu lassen, dem der Verkäufer und der Kunde vertrauen. Die Grundvariante dieses Ansatzes sich wie folgt aus:

1. Der Verkäufer (V) sendet dem Dritten (D) den Sourecode (S) in Verbindung mit der Testsuite und einer Beschreibung des gewünschten Überdeckungsgrades.
2. D generiert ein Binary aus S und konstruiert die Überdeckungsmenge.
3. D testet das Binary mit der Testsuite und überprüft, ob die Überdeckungsmenge mit der Testsuite übereinstimmt.
4. D teilt dem Kunden (K) mit, dass ein bestimmter Überdeckungsgrad durch die Testsuite erreicht wurde.
5. V sendet das Binary an K und teilt K mit, dass D den Überdeckungsgrad überprüft hat.

Dieser Ansatz hat eine Reihe von Einschränkungen wie zusätzliche Kosten, zeitliche Verzögerungen durch das Einbeziehen eines Dritten sowie die Schwierigkeit des Findens einer passenden, von beiden Seiten akzeptierten sogenannten Trusted-Third-Party. Wenn absolutes Vertrauen gefordert ist und eine vertrauenswürdiger Dritter gefunden wurde, verlieren die Argumente Kosten und zeitliche Verzögerung aber an Bedeutung.

Weitere Varianten dieses Ansatzes beschreiben das Testen durch Dritte wobei nur das Binary und nicht der Sourcecode an D übermittelt wird. Dabei wird im Bereich des Lieferanten ein Tool eingesetzt, das eine kryptografisch signierte Überdeckungsmenge generiert. Diese Überdeckungsmenge, das Binary und die Signatur werden an die Trusted-Third-Party übergeben, die die Signatur überprüft. Der weitere Ablauf entspricht dem oben beschriebenen. Für nähere Details und weitere Varianten sei auf [5] verwiesen.

5. Schlussfolgerung

Sowohl die (Wieder-)Verwendung von Software-Komponenten als auch das Testen sind bedeutende Bereich in der Softwareentwicklung zur Steigerung der Qualität. Dies wurde versucht in dieser Arbeit zu verbinden, vor allem aufgrund der besonderen Problemstellungen in der komponentenbasierten Softwareentwicklung. Nach einer kurzen Vorstellung des CBSE und der Definition einiger Begriffe wurden verschiedenste Ansätze zu diesem Thema vorgestellt. Das Repository-Design zielt auf inhouse-entwickelte Komponenten ab bzw. auf die Kenntnis des Quellcodes. Das Maturity-Model und der Ansatz zur Automatisierung holen weiter aus und beeinflussen nicht nur den Bereich des Testens. Hintergrund ist aber auch hier die Steigerung der Softwarequalität. Das effektive Testen ohne Kenntnis des Quellcodes ist eine große Herausforderung im CBSE. Dass auch Whitebox-Testen durch die Einschaltung eines Dritten möglich ist, ist ein vielversprechender Ansatz zur Entwicklung qualitativ hochwertiger Software.

6. Referenzen

[1] Y. Wu, D. Pan and M. Chen, "Testing Component-Based Software", Department of Information and Software Engineering, George Mason University, Fairfax, 2000.

[2] J. Gao, "Testing Component-based Software: Problems, Testability, and Maturity", Proceedings of Star'99, SQE, 1999.

[3] X. Cai, M. R. Lyu, K. F. Wong, "Component-Based Software Engineering: Technologies, Development Frameworks, and Quality Assurance Schemes", Asian-Pacific Software Engineering Conference (APSEC'2000), Singapore, December 2000.

[4] E. J. Weyuker, "Testing component-based software: A cautionary tale", IEEE Software, September/October 1998.

[5] P. T. Devanbu, S. G. Stubblebine, "Cryptographic Verification of Test Coverage Claims", IEEE Transactions on Software Engineering, Vol. 26, No. 2, February 2000.

[6] H. Balzert: Lehrbuch der Software-Technik - Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung, Berlin, 1998.

[7] J.M. Voas: Certifying Off-the-Shelf Software Components, Computer, Vol. 31, No. 6, 1998.

Effektives Testen Objektorientierter Software mit Objekt-Relations-Diagrammen

Stefan Perauer, Robert Sorschag
0061102, 0060423
{sperauer,rsorschl}@edu.uni-klu.ac.at

Abstract

In objektorientierten Systemen ist ein hoher Grad an Interaktion zwischen den Klassen vorhanden, wodurch das Testen erschwert wird. Ein großes Problem von Integrations- und Regressions-Testen ist es, eine effektive Testreihenfolge zu finden, die den nötigen Aufwand zum Testen minimiert. In dieser Arbeit werden Objekt-Relations-Diagramme (ORDs) vorgestellt, mit deren Hilfe dieses Problem gelöst werden kann.

Neben einer ausführlichen Beschreibung der Probleme des objektorientierten Softwaretests, wird ein Algorithmus zur Konstruktion beliebig präziser ORDs angeführt. Präzisionsunterschiede ergeben sich durch verschiedene Ansätze für ORDs, die entwickelt wurden. Weiters wird gezeigt, wie ORDs beim Integrations- und Regressions-Testen zum Einsatz kommen.

1. Einleitung

Objektorientierte Software besitzt einige grundlegende Unterschiede im Vergleich zu prozeduraler Software. Durch die Einführung neuer Konzepte wie Vererbung, Aggregation und Assoziation entstehen komplexe Abhängigkeiten zwischen den Klassen. Dadurch greifen traditionelle Testmethoden der prozeduralen Programmierung nicht, bei welchen die Komponenten erst einzeln getestet und anschließend in das Gesamtsystem integriert werden.

Bei objektorientierter Software werden als kleinste Test-Einheit die Klassen angesehen. Allerdings können Klassen aufgrund der Abhängigkeiten untereinander nicht gesondert getestet werden. Ein objektorientiertes System als Ganzes zu testen, wirft wiederum andere Probleme auf. Eine fehlerhafte Klasse kann andere Klassen destabilisieren. Der Fehler kann dann oft nicht mehr lokalisiert werden Ein anderes negatives Beispiel wären mehrere fehlerhafte Klassen, die zufällig ein richtiges Verhalten zeigen.

Deshalb wird beim Testen ein inkrementeller Ansatz bevorzugt. Zuerst werden Klassen getestet, die nicht von anderen Klassen abhängen. Im nächsten Schritt testet man die Klassen, die von schon getesten Klassen abhängen. Um eine Klasse zu testen, die von ungetesteten Klassen abhängt, muss zusätzlicher Aufwand betrieben werden. Ungetestete Klassen werden dabei durch *Stubs* simuliert. Da diese manuell erstellt werden, will man die Anzahl der nötigen Stubs minimieren. Daher ist die Testreihenfolge der Klassen wichtig (erst Klasse A testen, dann Klasse B, etc.). Schon getestete Klassen müssen nicht durch einen Stub simuliert werden.

Um eine Testreihenfolge zu finden, bieten sich Graphen an, die Klassen und ihre Abhängigkeiten darstellen. Ein Ansatz dafür sind die von Jéron et al. vorgestellten *Test-Dependency-Graphs* [6], die eine Erweiterung von UML-Klassendiagrammen sind. Diese haben den Nachteil, dass nicht erkennbar ist, welche Art der Abhängigkeit zwischen zwei Klassen vorliegt.

Ein anderer von Kung et al. vorgestellter Ansatz, der Klassenzusammenhänge explizit macht, sind *Objekt-Relations-Diagramme (ORDs)* [2]. In der vorliegenden Arbeit wollen wir ORDs vorstellen und zeigen, wie sie den Testprozess verbessern können. Alle angeführten Beispiele sind in Java geschrieben, allerdings sind die Konzepte nicht auf diese Sprache beschränkt.

2. Probleme des objektorientierten Testens

Durch die komplexen Klassenzusammenhänge in objektorientierten Programmen ist das Testen und die Wartung eine anspruchsvolle Aufgabe. Dafür gibt es folgende Gründe [2]:

- Es ist schwierig, Klassen in großen Programmen zu verstehen, wenn diese von vielen anderen Klassen abhängen.
- Ohne einen geeigneten Einblick in den Programmaufbau wissen Tester oft nicht, bei welchen Klassen der Testprozess begonnen werden soll.
- Das Schreiben von Stubs ist zeit- und damit kostenintensiv. Bei Experimenten wurde ermittelt, dass der Zeitaufwand für das Schreiben eines Stubs im Durchschnitt zwischen einer halben Stunde und einer Stunde liegt [1].
- Der Einfluss von Polymorphismus auf das Laufzeitverhalten ist schwer abzuschätzen.
- Die Auswirkungen von Programm-Änderungen pflanzen sich wegen Methoden-Aufrufketten durch das Programm fort. So eine Aufrufkette entsteht, wenn z.B. Klasse *A* eine Methode von Klasse *B* aufruft, welche wiederum die Methode einer anderen Klasse aufruft und so weiter. Kung et al. haben die Länge der Aufrufketten für die InterView C++ Grafikbibliothek mit 220 Klassen, mehr als 400 Klassen-Abhängigkeiten und insgesamt über 1000 Methoden, ermittelt (siehe Abbildung 1). Es ist ersichtlich, dass die meissten Aufrufketten eine Länge zwischen 2 und 7 besitzen.

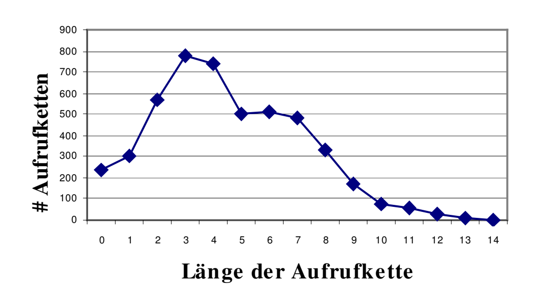


Abb.1: Aufrufketten der Interview Library

3. Objekt-Relations-Diagramme

ORDs wurden ursprünglich von Kung et al. entwickelt. Ein ORD ist ein gerichteter Graph, bei dem die Knoten Klassen darstellen und die Kanten Abhängigkeiten. Je nach Abhängigkeitstyp werden die Kanten mit *I* für Vererbung, *As* für Assoziation und *Ag* für Aggregation beschriftet. Der Graph kann zyklisch sein.

```
 1 class Parent{ public void meth(); }
 2 class Child extends Parent { public void meth(); }
 3 class Assoc{
 4   Parent par;
 5   public Assoc( Parent par ){
 6     this.par = par;
 7   }
 8   public void assocMeth(){
 9     this.par.meth();
10   }
11   public static void main(){
12     Child c  = new Child();
13     Assoc a = new Assoc(c);
14     a.assocMeth();
15   }
16 }
```

Abb.4: Programmbeispiel

Die ursprünglichen ORDs nach Kung [2] berücksichtigen nur statische Abhängigkeiten . Zur Laufzeit können aber durch Polymorphismus Abhängigkeiten entstehen, die in diesem ORD nicht aufscheinen. Ein weiteres Problem sind im ORD verzeichnete Abhängigkeiten, die zur Laufzeit nicht entstehen. Von Labiche et al. wurde ein verbesserter ORD vorgestellt, bei dem auch die dynamischen Abhängigkeiten berücksichtigt werden [7]. Allerdings wurde dabei das Problem überflüssiger Abhängigkeiten sogar noch verstärkt. Milanova et al. präsentierten eine Möglichkeit präzise ORDs zu konstruieren [1].

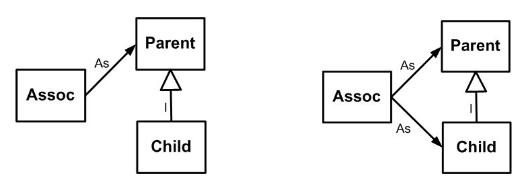


Abb.2: ORD nach Kung et al. bzw. Labiche et al.

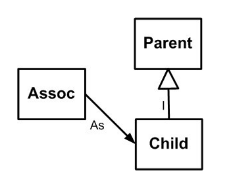


Abb.3: ORD nach Milanova et al.

Die Abbildungen 2 und 3 stellen die eben beschriebenen ORD-Varianten, die aus dem Java-Code von Abbildung 4 resultieren, dar. Kungs ORD berücksichtigt nur die statischen Abhängigkeiten

zwischen den 3 Klassen, die direkt aus dem Quelltext abgelesbar sind. Die Assoziationsbeziehung entsteht dabei, weil in der Klasse *Assoc* eine Methode eines im Konstruktor übergebenen *Parent*-Objekts aufgerufen wird. Labiche hat eine einfache Regel eingeführt, wonach jede Subklasse die Assoziationsbeziehungen seiner Superklasse erbt. Dies geht aus Abbildung 2 hervor. Die von Milanova et al. vorgestellten ORDs enthalten nur jene Abhängigkeiten, die zur Laufzeit tatsächlich auftreten (siehe Abbildung 3).

3.1 Klassenabhängigkeiten

Zunächst wollen wir Klassenabhängigkeiten, die aus Java-Quellcode gewonnen werden können, beschreiben [1]. Dabei wird die Notation *KA(A,B)* zur Beschreibung der Abhängigkeiten zwischen zwei Klassen *A* und *B* verwendet. *KA* steht hierbei für die Klassenabhängigkeit und ist aus der Menge *{I,As,Ag}*.

- **Vererbung:** Eine Vererbungsbeziehung *I(A,B)* zwischen zwei Klassen besteht, wenn sich in der Klassendefinition der Subklasse *A* das Schlüsselwort *extends B* befindet.
- **Aggregation:** Es existiert eine Abhängigkeit *Ag(A,B)* genau dann, wenn im Konstruktor von *A* ein Statement *this.var = new B()* existiert.
- **Assoziation:** Eine Abhängigkeit *As(A,B)* exisitiert genau dann, wenn entweder:
  - *m(..., B var, …)* die Definition einer Methode oder eines Konstruktors der Klasse *A* ist. Dies nennt man *Parameter Assoziation*.
  - Es exisitiert ein Variablenzugriff *b.var* in der Klasse *A* und die Variable *b* ist eine Instanz der Klasse *B*, wobei *b* keine Klassenvariable von *A* ist. Man nennt dies *Variablen-Zugriffs Assoziation.*
  - Die Klasse *A* enthält den Methodenaufruf *b.m(...)* und die Variable *b* ist eine Instanz der Klasse *B,* wobei *b* keine Klassenvariable von *A* ist. Man nennt dies *Methoden-Aufruf Assoziation.*

3.2 Generischer Algorithmus zur Konstruktion von ORDs

Die von Kung und Labiche vorgestellten Methoden zur Konstruktion von ORDs haben einen entscheidenden Nachteil: Sie sind ungenau. Dies führt zu überflüssigen bzw. fehlenden Kanten im ORD. Wir wollen einen generischen Algorithmus von Milanova et al. vorstellen, der auf einer vorhergehenden *Klassen-Analyse* basiert [1]. Der Vorteil dabei ist, dass die Genauigkeit des ORDs von der Genauigkeit der verwendeten Klassen-Analyse abhängt. Klassen-Analysen sind eine Form der statischen Programmanalyse und wurden ursprünglich für optimierende Compiler verwendet. Eine Klassen-Analyse liefert für jede Variable *x* eines Programms die Menge der Klassen *KM(x)*, welche zur Laufzeit möglicherweise an diese Variable gebunden werden. Die Klassen-Mengen sind je nach Analyse-Methode verschieden präzise.

Der Algorithmus in Abbildung 5 benötigt als Eingabe alle Statements und Methoden eines Programms, sowie die Ergebnisse der Klassen-Analyse des selben Programms. Das Resultat des Algorithmus ist ein ORD, welcher je nach Präzisiongrad der vorhergehenden Klassenanalyse in der Genauigkeit variiert. Der damit gewonnene ORD unterscheidet sich nur in den Assoziations-Beziehungen vom ORD des naiven Ansatzes.

EINGABE:
Statements: Menge von Statements
Methoden: Menge von Methoden
KM : Ausgabe der Klassenanalyse

AUSGABE: ORD
// *Vererbungsbeziehung*
Für jede direkte Subklasse *S* der Klasse *K*:
    ORD := ORD ∪ I(S,K)
// *Aggregationsbeziehung*
Für jedes Statement *s* der Form *this.var = new K(…)*
in einem Konstruktor der Klasse *A*:
        ORD := ORD ∪ Ag(A,K)
// *Variablen-Zugriffs-Assoziationsbeziehung*
Für jedes Statement *s* der Form *obj.var* in der Klasse *A*
  Wenn o≠this :
    ORD := ORD ∪ As (A, KM(obj))
// *Methoden-Aufruf-Assoziationsbeziehung*
Für jeden Methodenaufruf *obj.m(..)* in der Klasse *A*
  Wenn o≠this :
    ORD := ORD ∪ As (A, KM(obj))
// *Parameter-Assoziations*beziehung
Für jede Methode *m* der Klasse *A*:
  Für jeden Parameter p der Methode m:
        ORD := ORD ∪ As (A, KM(p))

Abb.5: Generischer Algorithmus für ORDs

Präzise ORDs haben mehrere Vorteile. Da sie nur die tatsächlich auftretenden Abhängigkeiten enthalten, werden beim Testen auch nur diese berücksichtigt. Durch unpräzise ORDs kommt es zu überflüssigen bzw. fehlenden Testaufwand. Im allgemeinen besitzen präzise ORDs bis zu 40% weniger Abhängigkeiten, wodurch weniger Zyklen entstehen und so weitere Probleme vermieden werden (siehe Abschnitt 4).

3.3 Extended-ORDs

Extended ORDs (Ext-ORDs) sind eine Erweiterung der schon beschriebenen ORDs. Dabei werden Aggregations- und Assoziations-Kanten mit Zusatzinformationen beschriftet, die aussagen, welches Statement eine Abhängigkeit auslöst. Zwischen zwei Klassen können so mehrere Kanten entstehen. In Abbildung 6 ist ein Beispiel für einen Ext-ORD angegeben, wobei aus Platzgründen kein Quelltext angegeben ist. Die Zahlen hinter dem Doppelpunkt der Abhängigkeit stehen für die Zeilennummer in der diese ausgelöst wurde.

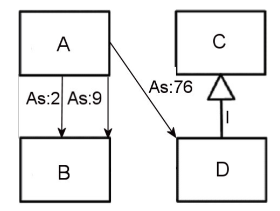


Abb.6: Ext-ORD

Der in Abbildung 5 beschriebene Algorithmus kann mit Hilfe von 2 Regeln zur Konstruktion von Ext-ORDs erweitert werden:

- Es exisitiert eine Kante *Ag:*s_i, wenn Statement s_i eine Aggregationsbeziehung auslöst.
- Es existiert eine Kante *As:*s_i, wenn Statement s_i eine *Variablen-Zugriffs-Assoziation* oder eine *Methoden-Aufruf-Assoziation* auslöst.

4. Effektives Testen mit ORDs

Wie schon erwähnt, ist ein Hauptproblem beim Testen objektorientierter Software eine geeignete Testreihenfolge für die Klassen zu finden. Eine geeignete Metrik für eine Testreihenfolge stellen die benötigen Stubs dar. Ein Stub muss immer dann erzeugt werden, wenn eine zu testende Klasse von ungetesteten Klassen abhängt. Stubs stellen die minimale Funktionalität bereit, die für die Klasse im Test notwendig ist. Sie sollten möglichts einfach gehalten sein und nur aus einer sequenziellen Ausführung bestehen, um selbst keine Fehler zu enthalten [5,6].

Außerdem muss man berücksichtigen, dass eine Klasse die von mehreren Klienten-Klassen benutzt wird, nicht nur durch einen Stub simuliert werden kann. Für jede Klient-Klasse muss ein eigener Stub geschrieben werden, da jeder Stub nur ein spezifisches Verhalten für eine Klasse bereitstellt.

Bei der Verwendung von ORDs zur Ermittlung einer Testreihenfolge ergibt sich das Problem von Zyklen. Wenn ein Zyklus vorhanden ist, müssen Abhängigkeitskanten gelöscht werden, um diesen aufzulösen. Für jene Klasse, auf welche die gelöschte Kante gerichtet war, muss in weiterer Folge ein Stub geschrieben werden. Aus folgenden Gründen ist es am sinnvollsten, ausschließlich Assoziationsbeziehungen zu löschen [4]:

- Aggregationsbeziehungen erfordern in der Regel mehrere Stubs, wenn sie gelöscht werden.
- Bei Vererbung übernehmen Kindklassen mehrere Methoden ihrer Elternklassen. Wenn man eine Elternklasse stubben möchte, muss man für fast jede Elternmethode einen Stub schreiben.
- Assoziationsbeziehungen erfordern beim Löschen hingegen meist nur einen sehr spezifischen Stub und stellen so die schwächsten Abhängigkeiten zwischen Klassen dar.

Es gibt mehrere Algorithmen zum Auflösen von Zyklen, welche qualitativ unterschiedlich sind. Die ursprünglich von Kung et al. vorgeschlagene Methode [2] wählt zufällig eine zu löschende Assoziationsbeziehung, was ein eher naiver Ansatz ist. Wir wollen deshalb den Algorithmus von Briand et al. [5] vorstellen, der sehr gute Resultate bezüglich der erforderlichen Stub-Anzahl erzielt.

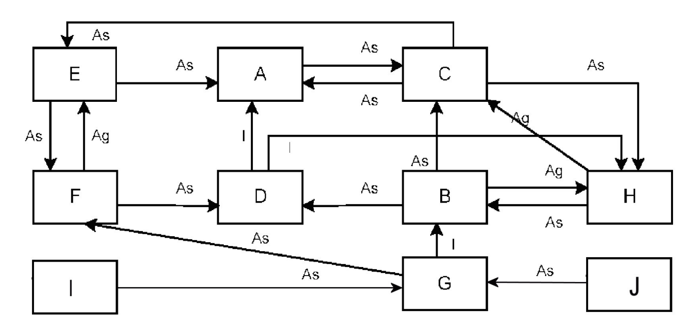


Abb.7: Zyklischer ORD

4.1 Auflösen von Zyklen nach Briand et al.

Bei diesem Algorithmus werden rekursiv, so genannte *Strongly-Connected-Components (SCCs)* ermittelt und von diesen die am besten geeignetste Assoziationsbeziehung gelöscht. Eine SCC ist ein Subgraph eines ORDs, wobei für alle Knoten *u* und *v* der SCC gilt: *u* ist durch einen Weg mit *v* verbunden und umgekehrt. SCCs werden durch den Algorithmus von Tarjan [6] gefunden.

Innerhalb einer SCC werden alle darin vorhanden Assoziationsbeziehungen mit der Formel

$$As(A,B) = B_{in} * A_{out}$$

gewichtet. B_{in} bezeichnet die Anzahl der eingehenden Kanten der Klasse *B*, A_{out} die Anzahl der ausgehenden Kanten der Klasse *A*. Die Kante mit dem höchsten Wert wird gelöscht. Gibt es mehrere Kanten mit gleichem Maximalwert, wird eine davon zufällig ausgewählt.

Danach werden wiederholt SCCs ermittelt und die am stärksten gewichtete Assoziationsbeziehungen gelöscht, bis keine Zyklen mehr vorhanden sind. Eine Testreihenfolge kann nun durch eine topologische Sortierung auf dem azyklischen Graphen ermittelt werden.

4.1.1 Beispiel

Zuerst wird der Algorithmus von Tarjan angewendet, um im gegebenen ORD aus Abbildung 7 eine SCC zu finden. Das Resultat ist die SCC_1 *: {F, E, C, A, D, B, H}*. Alle darin vorhandenen Elemente sind transitiv, zyklisch miteinander verbunden. Nur die Klassen *G*, *I* und *J* gehören der SCC_1 nicht an. Von der so gewonnenen SCC_1 muss nun eine Kante gelöscht werden, um einen Zyklus aufzulösen. Dafür werden alle Kanten der SCC_1 gewichtet (siehe Tabelle 1) und eine Kante mit maximalem Gewicht wird gelöscht. In diesem Fall kommen dafür die Kanten *As(H,B)* und *As(A,C)* in Frage.

As(H,B)	$H_{in} * B_{out}$	3 * 3	9
As(A,C)	$A_{in} * C_{out}$	3 * 3	9
As(E,A)	$E_{in} * A_{out}$	2 * 1	2
As(C,H)	$C_{in} * H_{out}$	3 * 2	6
As(B,D)	$B_{in} * D_{out}$	1 * 2	2
As(C,A)	$C_{in} * A_{out}$	3 * 1	3
As(E,F)	$E_{in} * F_{out}$	2 * 2	4
As(B,C)	$B_{in} * C_{out}$	1 * 3	3
As(C,E)	$C_{in} * E_{out}$	3 * 2	6
As(F,D)	$F_{in} * D_{out}$	1 * 2	2

Tab.1: Kantengewichtung der SCC_1

Wir entfernen die Kante *As(A,C).* Als nächster Schritt wird aus der SCC_1 mit Hilfe des Algorithmus von Tarjan die SCC_2*: {F, E, C, D, B, H}* erzeugt. Die darin enthalten Kanten werden wieder gewichtet und eine Maximalwert-Kante, in diesem Fall *As(H,B)*, wird gelöscht. Im Folgenden werden noch SCC_3 und SCC_4 erzeugt, wobei die Kanten *As(E,F)* und *As(C,H)* gelöscht werden. In Abbildung 8 sind alle Schritte dargestellt. Die verbleibende SCC_4 enthält nach dem Löschen der *As(C,H)* keine Zyklen mehr und der Algorithmus terminiert.

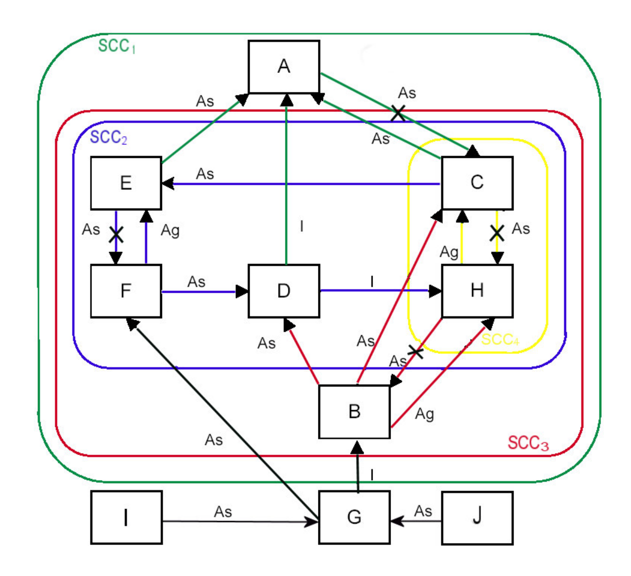


Abb.8: SCCs und gelöschte Kanten des ORDs

4.2 Integrations-Testen

Das Ziel des Integrations-Testen ist, Fehler aufzudecken, die durch Klasseninteraktionen zustande kommen. Dabei ist immer eine Zielmenge von Klassen im Test. Alle Klassen die nicht in der Zielmenge liegen, müssen durch Stubs simuliert werden. Für die Klassen der Zielmenge muss wiederum eine Testreihenfolge gefunden werden. Dafür müssen vorher alle Zyklen aufgelöst werden.

Für das eben beschriebene Beispiel zeigen wir folgend eine Integrations-Testreihenfolge, wobei wir annehmen, dass alle Klassen in der Zielmenge liegen. Da alle Zyklen bereits entfernt wurden, kann die Testreihenfolge durch eine topologische Sortierung auf dem azyklischen ORD ermittelt werden:

1. *A* wird mit dem *Stub C* getestet
2. *E* wird mit *A* und dem *Stub F* getestet
3. *C* wird mit *A*, *E* und dem *Stub H* getestet
4. *H* wird mit *C* und dem *Stub B* getestet
5. *D* wird mit *A* und *H* getestet
6. *F* wird mit *E* und *D* getestet
7. *B* wird mit *C*, *D*, und *H* getestet
8. *G* wird mit *F* und *B* getestet
9. *I* wird mit *G* getestet
10. *J* wird mit *G* getestet

Dabei werden genau 4 Stubs benötigt. Für dieses kleine Beispiel kann gezeigt werden, dass dieses Resultat optimal ist.

4.2.1 Integrations-Abdeckungsanalyse

Durch die Integrations-Abdeckungsanalyse ist die Bewertung eines Testprozesses möglich. Dabei wird evaluiert, wieviele Klasseninteraktionen getestet wurden. So können konkrete Qualitätsanforderungen an einen Integrations-Test gestellt werden, da im Allgemeinen nicht alle Klasseninteraktionen getestet werden. Ein mögliches Kriterium wäre, zumindest 80% aller Variablen- und Methoden-Aufruf-Assoziationen zu testen. Hierbei ist der bereits vorgestellte Ext-ORD hilfreich, da dieser für jedes Statement, welches eine Assoziationsbeziehung auslöst, eine Kante enthält.

4.3 Regressions-Testen

Regressions-Testen ist das erneute Testen eines Programms, nachdem man Änderungen vorgenommen hat. Der Regressions-Test soll aufzeigen, ob das Programm nach der Änderung noch den Anforderungen entspricht und keine neuen Fehler enstanden sind. Natürlich müssen nur die Teile bzw. Klassen erneut getestet werden, die auch von der Änderung betroffen sind [3]. Beim Regressions-Testen gibt es 4 grundsätzliche Probleme, die teilweise mit ORDs gelöst werden können:

1. Wie identifiziert man die Komponenten, die von einer Änderung betroffen sind?
2. Welche Strategie sollte angewandt werden, um die betroffenen Klassen zu testen, bzw. in welcher Reihenfolge sollten diese getestet werden?
3. Ab welchem Test-Abdeckungsgrad ist genügend getestet worden?
4. Wie können bereits vorhandene Testfälle für den Regressions-Test wiederverwendet werden?

Durch die Abbildung der Klassenabhängigkeiten in einem ORD kann man leicht herausfinden, welche Klassen von einer Änderung betroffen sind. In diesem Zusammenhang wurde von Kung et al. das Konzept der Class-Firewall (CFW) vorgestellt. $CFW(X)$, die Class-Firewall der Klasse X, ist wie folgt definiert:

$$CFW(K) = \{ X_i \mid (X_i, K) \in E^* \}$$

E^* ist die transitive Hülle aller Kanten E des ORDs. Wenn die Kante (X_1, X_2) und (X_2, X_3) in E enthalten sind, so ist neben diesen beiden Kanten auch die Kante (X_1, X_3) in E^* enthalten. $CFW(K)$ enthält also alle von K transitiv abhängigen Klassen.

Beispiel: Die Klasse G aus Abbildung 7 wurde geändert und es soll ein Regressionstest gemacht werden. Die Class-Firewall von G lautet:

$$CFW(G)=\{ I , J \}$$

Nur die Klassen I, J sind transitiv von G abhängig und müssen erneut getestet werden. Falls I und J zyklisch abhängig wären, müsste ein Algorithmus zum Auflösen des Zyklus angewandt werden, um eine Testreihenfolge zu finden (siehe Abschnitt 4.1). So ergibt sich die Reihenfolge G vor I und J.

5. Zusammenfassung

In dieser Arbeit wurde die Nützlichkeit von ORDs für den Testprozess, speziell für Integrations- und Regressions-Testen, gezeigt. Die Verwendung von ORDs zur Ermittlung einer Testreihenfolge kann im Vergleich zu einer zufälligen Reihenfolge bis zu 90% Einsparung an Stubs bringen. Durch die Konstruktion präziser ORDs und einem optimalen Algorithmus zum Auflösen von Zyklen kann der Testprozess zusätzlich verbessert werden.

6. Referenzen

[1] Ana Milanova, Atanas Rountev, and Barbara G. Ryder, "Constructing Precise Object Relation Diagrams", *Proceedings of the International Conference on Software Maintenance,* October, 2002

[2] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen, ``A test strategy for object-oriented systems", *Proc. of Computer Software and Applications Conference*, pp. 239 -- 244, *IEEE Computer Society*, 1995

[3] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen, ``Class firewall, regression testing, and software maintenance of object oriented systems", *Journal of Object Oriented Programming*, pp. 51 -- 65, May 1995

[4] Brian A. Malloy, Peter J. Clarke and Errol L. Lloyd, "A Parameterized Cost Model to Order Classes for Class-based Testing of C++ Applications", *Proceedings of International Symposium on Software Reliability Engineering (ISSRE'03),* Denver Colorado, November 17-20, 2003, pages 353-364.

[5] L. Briand, Y. Labiche, Y. Wang, "An Investigation of Graph-Based Class Integration Test Order Strategies", *IEEE Transactions on Software Engineering, vol. 29 (6)*, 2003

[6] Thiéry Jéron, Jean-Marc Jézéquel, Yves Le Traon, and Pierre Morel, "Efficient Strategies for Integration and Regression Testing of OO Systems", *Proc. of the 10th International Symposium on Software Reliability Engineering (ISSRE'99)*, November 1999

[7] Y. Labiche, P. Thévenod-Fosse, H. Waeselynck and M.-H. Durand, "Testing Levels for Object-Oriented Software", *Proc. 22nd IEEE International Conference on Software Engineering (ICSE)*, Limerick (Ireland), June, 2000.

Mutationentest - Objektorientierte Mutanten für Java-Programme

Innerwinkler Daniela, Gunar Mätzler
0060550, 0060396
{dinnerwi, gmaetzle}@edu.uni-klu.ac.at

Abstract. *Der Mutationentest ist eine auf Fehler basierende Testmethode, mit deren Hilfe die Effektivität von Testdaten bewertet werden kann. Der traditionelle Test kann bestimmte Arten von Fehlern, die in objektorientierten Programmen auftreten, übersehen. Die Ursache dafür ist die Tatsache, dass Mutationsoperatoren aus Studien mit nicht-objektorientierten Programmiersprachen wie Fortran oder C abgeleitet wurden und somit nur die Merkmale prozeduraler Programmiersprachen unterstützen. In der folgenden Abhandlung wird anhand der Programmiersprache Java auf spezifische Fehler eingegangen, die durch die Merkmale objektorientierter Sprachen entstehen können und betrachtet, wie bestehende Ansätze durch die Einführung neuer Mutationsoperatoren zu passenden Mutanten führen können. Die Frage inwieweit marktreife Tools für objektorientiertes Mutationentesten verfügbar sind und ob dieser Ansatz den Sprung aus der Theorie in die Praxis geschafft hat, wird ebenfalls behandelt.*

1. Einleitung

Der Mutationentest ist eine sehr leistungsfähige Testmethode, um adäquate Testdaten generieren und bewerten zu können. In der Praxis wird diese Art des Unit-Tests nur spärlich eingesetzt, da der Aufwand enorm groß und nur für entsprechend sensible Bereiche wie Life-Critical-Systems tragbar ist. 1971 wurde die Idee des auf Fehler basierenden Tests von Richard Lipton in seiner Arbeit „Fault Diagnosis of Computer Programms“ veröffentlicht. Wegweisend für den Mutationentest war die Publikation „Hints on test data selection: Help for the practicing programmer“ von DeMillo, Lipton und Sayward. Schon früh wurde dieses Konzept in einem entsprechenden Werkzeug für prozedurale Programmiersprachen umgesetzt (Mothra). Doch wie sieht es mit Tools für objektorientierte Sprachen wie Java aus, die immer mehr an Bedeutung gewinnen? Bevor diese Frage beantwortet wird möchten wir kurz das Konzept des Mutationentest erläutern.

2. Mutationentest

Dieser Test prüft, ob die Testdaten zwischen dem ursprünglichen Programm und den davon abgewandelten Programmen oder Programmteilen, die für spezifische Fehlertypen stehen, unterscheiden können. Dadurch wird die Effektivität der Testdaten ermittelt, aber auch die im Programm selbst vorkommenden Fehler aufgezeigt. In der IEEE Standard Terminologie werden Fehler in „Failures“ und „Faults“ eingeteilt. „Failures“ beschreiben externe, ungültige Zustände eines Programms (z.B. Laufzeitfehler oder ungültige Ausgaben). „Faults“ hingegen sind interne, ungültige oder fehlerhafte Programmzeilen, die mittels Mutationentest erkennbar gemacht werden.[1]

Zur Erstellung der abgewandelten Programme, auch Mutanten genannt, dienen sogenannte Mutationsoperatoren. Ein solcher Operator ist eine Regel, die beschreibt, wie ein Konstrukt oder Operand einer Sprache in ein syntaktisch legales, modifiziertes oder sogar gelöschtes Element umgewandelt wird. Mittels dieser Operatoren können durch ein Mutationssystem automatisch Mutanten erzeugt werden.[1] Dabei gilt die Annahme, dass einfache Fehler durch Kopplungseffekte zu komplexeren Fehlern führen und erkannt werden.[2]

Der Ablauf des Mutationentests stellt sich wie folgt dar: Zuerst wird das Originalprogramm mit der Menge T getestet. Tritt ein Fehler auf, wird das Programm verbessert. Erkennt die Testmenge T im Originalprogramm keinen Fehler, werden Mutanten erzeugt. Durch Anwendung von Mutationsoperatoren entsteht jeweils ein neuer Mutant M_i der einen Fehler F_j enthält. Kann nun das Mutationssystem durch Anwendung der Testmenge T den Fehler F_j erkennen,

dann gilt dieser Mutant als „killed“ und ist somit nicht mehr von weiterer Relevanz. F_j kann aus zwei Gründen unentdeckt bleiben: 1. der Mutant ist funktional äquivalent („equivalent mutation“) oder 2. die Testmenge konnte den Fehler nicht finden, obwohl der Mutant nicht-äquivalent ist („killable“, „non-equivalent“).[2]

Ein Mutant ist funktional äquivalent zum Original, wenn er für alle Inputmengen den gleichen Output erzeugt bzw. es keinen Testfall gibt, der ihn erkennen kann. Wenn ein Mutant nicht-äquivalent ist, so existiert eine Testmenge T’, welche den Fehler F_j erkennt. Somit gilt, dass die bisherige Menge T ungenügend war. Die Testmenge T muss solange erweitert werden, bis sie „strong“ ist, d.h. dem Tester genügt und F_j erkennt.[3]

Eine Aussage über die Effektivität der Testdaten ermöglicht der „Mutation-Score“. Dieser Wert gibt Aufschluss über die Anzahl der erkannten Mutanten. Er ist definiert als die (Anzahl der erkannten Mutanten)/ (Anzahl der nicht-äquivalenten Mutanten) * 100. Beträgt der Mutation-Score 100 %, so ist die Testmenge mutations-adäquat („mutation-adequate“) d.h. sie konnte alle Fehler finden.[4]

Aufgrund des enormen Aufwands der Erzeugung und Ausführung von Mutanten sowie der Erkennung von äquivalenten Mutanten wurden verschiedene Varianten des Mutationentests wie „selective Mutation“ und „weak Mutation“ entwickelt. Der zuvor beschriebene, traditionelle Ablauf des Mutationentests ist aus Figure 1 ersichtlich. Die mit gestrichelter Linie umrandeten Aktivitäten sind dabei von Hand auszuführen. Der Aufwand beläuft sich auf $O(N^2)$, wobei N die Anzahl der Referenzen in einem Programm darstellt.[4]

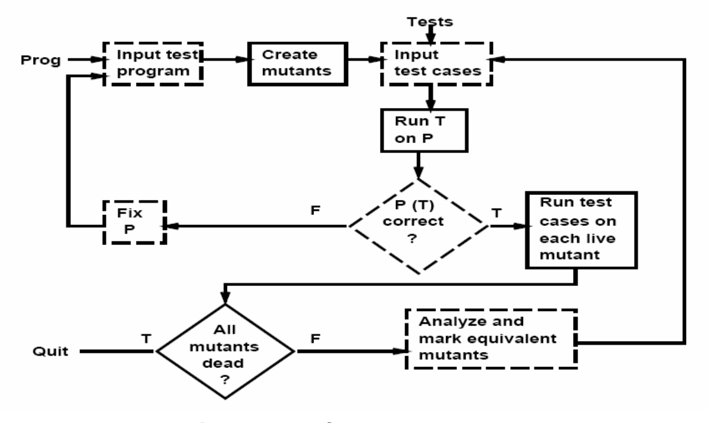


Figure 1. Ablauf Mutationentestprozess[2]

3. Objektorientiertes Mutationentesten

Im Folgenden wird anhand der Programmiersprache Java auf die wichtigsten Merkmale objektorientierter Sprachen eingegangen. Dieser Abschnitt dient als Grundlage für die weitere Entwicklung der Arbeit, da Mutationsoperatoren auf Fehlern basieren, die unter anderem durch Sprachmerkmale bedingt sind.

3.1 Objektorientierte Sprachmerkmale (Java)

Kapselung

Kapselung ist ein Verfahrung zur Minimierung von gegenseitigen Abhängigkeiten zwischen getrennt geschriebenen Modulen durch strikte, externe Interfaces.[5]

Durch Kapselung ist es Objekten möglich, den Zugriff auf ihre Variablen und Methoden durch andere Objekte einzuschränken. In Java wurde dieses Konzept durch vier verschiedene Zugriffsmodifikatoren verwirklicht.[6]

Zugriffsmodifikatoren in Java[6]

Modifi-kator	Gleiche Klasse	Subklasse im selben Package	Selbes Package, keine Subklasse	Anderes Package, Subklasse	Anderes Package, keine Subklasse
Private	Ja	Nein	Nein	Nein	Nein
Default/ Package	Ja	Ja	Ja	Nein	Nein
Protected	Ja	Ja	Ja	Ja	Nein
Public	Ja	Ja	Ja	Ja	Ja

Vererbung

„Vererbung ist ein Mechanismus, der es ermöglicht, Eigenschaften und Fähigkeiten von Klassen an andere Klassen weiterzugeben.“[7]

Java unterstützt keine Mehrfachvererbung. Die Subklasse erbt die Variablen und Methoden ihrer Elternklasse und aller anderen Vorfahren. Jede Klasse besitzt nur eine direkte Elternklasse. Die Subklasse kann die geerbten Methoden und Variablen mithilfe des Schlüsselwortes „super“ aufrufen. Die Umsetzung der Vererbung in Java erlaubt das Überschreiben von Methoden, das Verstecken von Variablen und Klassenkonstruktoren.

Überschreiben von Methoden

Die Subklasse kann Methoden der Superklasse neu definieren. Die Methode der Subklasse hat dieselbe Signatur aber eine unterschiedliche Implementierung.

Verstecken von Variablen

Java bietet die Möglichkeit in der Subklasse dieselbe Variable (Typ, Name) wie in der Superklasse zu definieren. Das hat den Effekt, dass die Variable der Superklasse vor der Subklasse „versteckt" ist.

Klassenkonstruktoren

Um den Konstruktor der Superklasse zu verwenden, muss ein expliziter Aufruf mit „super" erfolgen. Dieser Aufruf muss die erste Anweisung im Konstruktor der Subklasse sein. Ist diese Anweisung nicht vorhanden, wird sie vom Java-Compiler implizit eingefügt und der Default-Konstruktor der Superklasse ausgeführt.[6]

Polymorphismus

„Unter Polymorphismus ist die Fähigkeit einer Größe (Objekte und somit implizit die mit ihnen verbundenen Methoden, Funktionen, Variablen, etc.) zu verstehen, zur Laufzeit unterschiedliche Ausprägungen bzw. Formen annehmen zu können."[7]

Java unterstützt Polymorphismus für Attribute und Methoden, beides mithilfe der dynamischen Bindung. Jedes Objekt hat einen deklarierten Typ und einen aktuellen Typ. Der aktuelle Typ kann vom deklarierten Typ erben. Ein polymorphes Attribut ist eine Objektreferenz, die unterschiedliche Typen annehmen kann. Eine polymorphe Methode kann Parameter unterschiedlichen Typs akzeptieren. Dies ist möglich, wenn sie einen Parameter vom Typ „Object" besitzt.[6]

Überladen

Werden für Konstruktoren und Methoden dieselben Namen aber unterschiedliche Signaturen verwendet, spricht man vom Überladen.[6]

Spezifische Merkmale von Java

STATIC

Wird eine Konstante oder Variable als statisch deklariert, existiert nur eine Ausprägung davon, egal wie viele Instanzen einer Klasse erzeugt werden. Die Ausprägung der Klassenvariablen wird bei der Initialisierung der Klasse erzeugt. Im Gegensatz dazu stehen die Instanzvariablen (non-static). Jedes Mal, wenn eine neue Instanz der Klasse erzeugt wird, wird eine damit in Beziehung stehende Ausprägung der Instanzvariablen erzeugt. Eine statische Methode (Klassenmethode) wird ohne Referenz auf ein bestimmtes Objekt aufgerufen.[8]

THIS

Das Schlüsselwort „this" bezeichnet einen Wert, der eine Referenz auf das aktuelle Objekt darstellt. Über diese Referenz können (überschriebene) Instanzmethoden oder Instanzvariablen aufgerufen werden. Der deklarierte Typ von „this" ist die Klasse, in der das Schlüsselwort vorkommt. Der aktuelle Typ kann diese Klasse oder eine ihrer Subklassen sein. „this" wird auch beim expliziten Aufruf eines Konstruktors der Superklasse als erste Anweisung im Konstruktor der Subklasse verwendet.[8]

3.2 Mutationsoperatoren für Java

Die im Folgenden erwähnten Mutationsoperatoren stützen sich primär auf den Artikel „InterClass Mutation Operators for Java".[6] Diese Aufzählung ist nicht umfassend, jedoch beinhaltet sie wichtige Mutationsoperatoren für häufige Programmierfehler, die im Zusammenhang mit den vorhergehend erwähnten objektorientierten Sprachmerkmalen stehen.

Kapselung

AMC – Access modifier change

Dieser Operator ändert die Zugriffsmodifikatoren public, private und protected in deren Alternativen.[6] Dadurch soll der korrekte Zugriff sichergestellt und dem Paradigma der Kapselung entsprochen werden.[9] Dieser Fehler wird erkannt, wenn durch die Änderung Zugriffsrechte eingeschränkt werden bzw. Namenskonflikte entstehen.[6]

Vererbung

Vererbung hat unter anderem den Vorteil der realitätsnahen Abbildung der bestehenden Welt. Der Leistungsfähigkeit dieses Sprachkonstrukts steht jedoch die „große Gefahr der Unüberschaubarkeit und daraus resultierender unerwünschter Nebeneffekte"[7] und deren Fernwirkungen bei Wartungsarbeiten gegenüber.[7] Die folgenden Mutationsoperatoren sollen diese unerwünschten Fehlerquellen aufzeigen:

IHD – Hiding variable deletion

Dieser Mutationsoperator simuliert den typischen Fehler, dass ungewollt auf eine falsche, namensgleiche Instanzvariable zugegriffen wird.[9] Dazu wird in einer Subklasse die namensgleiche Instanz gelöscht.[6]

IHI –Hiding variable insertion

Mit der selben Absicht, wie beim IHD, führt IHI neue überschreibende Variablen ein. Diese können nur von Testfällen erkannt werden, die falsche Referenzen auf überschreibende Variablen erkennen[6] bzw. es muss zumindest einmal auf die entsprechende Variable zugegriffen werden, da dieser Fehler sonst unentdeckt bleibt.[9]

IOD – Overriding method deletion

Um den Aufruf der richtigen Methode in der Subklasse sicherzustellen, wird die überschreibende Methode der Subklasse entfernt und dadurch die Methode der Elternklasse ausgeführt.[6] Kann diese Änderung nicht erkannt werden, ist die Testmenge nicht adäquat oder die überschreibende Methode hatte die gleiche Funktionalität.[9]

IOP - Overridden method calling position change

Manchmal ist es notwendig, die Methode der Superklasse in der Methode der Subklasse, die sie überschreibt, aufzurufen. Dies kann dazu führen, dass der Aufruf unbeabsichtigt zum falschen Zeitpunkt erfolgt. IOP simuliert diesen Fehler durch das Vertauschen der Reihenfolge der Anweisungen.[6]

IOR – Overridden method rename

IOR dient zur Überprüfung von Abhängigkeiten überschriebener Methoden, die zu unerwünschten Nebeneffekten führen können. Zum Beispiel ruft eine Methode m() eine Methode f() der selben Klasse auf. Eine Klasse erbt nun diese Methoden und überschreibt f(). Der Aufruf von m() führt dazu, dass die Methode f() in der Subklasse aufgerufen wird, was ungewollt zu Fehlern führen kann. Zur Überprüfung dessen wird die Methode der Superklasse umbenannt.[6]

Original Code		IOR Mutant
Class List {		Class List {
void m() { .. f();..}	Δ	void m() { .. f'();..}
void f() {…}	Δ	void f'() {…}
}		}
class Stack extends List {		class Stack extends List {
void f() {…}		void f() {…}
}		}

Figure 2. IOR[6]

ISK – "super" keyword deletion

Dieser Operator entfernt den Aufruf von überschriebenen Methoden bzw. versteckten Variablen mittels "super". Die Referenz wird folglich aus der Subklasse entfernt und dadurch sichergestellt, dass Methoden und Variablen zweckmäßig verwendet werden.[6]

IPC – Explicit call of a parent's consturctor deletion

Bei der Erzeugung eines Objektes einer Subklasse, wird automatisch der argumentlose Default-Konstruktor der Superklasse ausgeführt. Das kann durch den Aufruf eines speziellen Konstruktors der Superklasse mittels „super" umgangen werden. Diese Anweisungen werden entfernt.[6]

Polymorphismus

Polymorphismus bietet eine sehr große Flexibilität, die jedoch mit großer Verantwortung einhergeht, da der Programmierer wissen sollte, welche Ausprägung eine Größe zur Laufzeit haben kann.[7] Die korrekte Anwendung dieses objektorientierten Konzepts soll durch folgende Mutationsoperatoren sichergestellt werden:

PNC – "new" method call with child class type

PNC ersetzt den durch den Aufruf von „new" zugewiesenen Typ einer Objektreferenz mit einem kompatiblen Typ (Subklasse).[6] Dadurch wird das Verhalten von kompatiblen Typen aufgezeigt und unter Umständen auch mögliche ungewollte Typverwendungen aufgedeckt.[9] Zum Beispiel (siehe Figure 3) wird eine Objektreferenz mit dem Subtyp B einer Klasse statt dem eigentlichen Supertyp A erzeugt.

Original Code		PNC Mutant
A a;		A a;
a = new A();	Δ	a = new B();

Figure 3. PNC[6]

PMD – Member variable declaration with parent class

Im Gegensatz zu PNC ändert PMD die Deklaration eines Typs zu dessen Elternklasse, d.h. nach obigen Beispiel würde a als C deklariert werden. C ist Elternklasse von A. Um diesen Fehler zu erkennen, muss die Testmenge ein ungültiges Verhalten aufgrund dieses Typs erzeugen.[6]

PPD – Parameter variable declaration with child class type

PPD entspricht dem Mutationsoperator PMD. PPD arbeitet jedoch auf Parameterreferenzen von Methoden. Er ändert Parameterreferenzen auf mögliche Elternklassen.[6]

PRV – Reference assignment with other comatible

PRV ersetzt Objektreferenzzuweisungen durch kompatible Objekte.[6] Er entspricht insoweit PNC, ist aber mächtiger.

Überladen

Überladen hat zur Folge, dass der Programmierer genau wissen muss, welche Methode er wann mit welchen Parametern aufrufen sollte. Die folgenden Mutationsoperatoren verdeutlichen diesen Aspekt:

OMR – Overloading method contents change

OMR überprüft den zweckmäßigen Aufruf von überladenen Methoden. Dabei werden Methodenrümpfe gleichnamiger Methoden mit unterschiedlichen Signaturen vertauscht, d.h. mittels „this“ wird in einer überladenen Methode eine namensgleiche Methode aufgerufen.[6]

OMD – Overloading method deletion

Dieser Mutationsoperator löscht überladene Methodendeklarationen. Sollte der Mutant trotzdem korrekt arbeiten, kann dies auf eine inkorrekte Verwendung der überladenen Methode hinweisen.[6] Außerdem kann die Nicht-Verwendung von Methoden aufgedeckt werden.[9]

OAO – Argument order change1

OAO vertauscht die Parameter von Methodenaufrufen überladender Methoden, falls kompatible Methodensignaturen vorhanden sind. Er überprüft die richtige Verwendung von Methodenaufrufen, die durch ungewollte Parametertypkonvertierungen für den Programmierer richtig erschienen sind.[9]

OAN – Argument number change

Ändert die Anzahl der Argumente bei den Aufrufen von überladenen Methoden, wenn eine Methode vorhanden ist, die die neue Argumentliste akzeptiert. Dies untermauert die Richtigkeit des Methodenaufrufes.[6]

Spezifische Merkmale von Java

JTD – “this” keyword deletion

JTD löscht die Verwendung des Schlüsselwortes “this”, welches erlaubt, versteckte Instanzvariablen in einer Methode zu referenzieren. Dadurch stellt dieser Operator den richtigen Gebrauch von Zustandsvariablen sicher.[6]

JSC – “static” modifier change

Ändert den “static”-Modifikator von Klassenvariablen zu Instanzvariablen oder umgekehrt. Dies ermöglicht das Verhalten von Instanz- bzw. Klassenvariablen festzustellen.[6]

JID – Member variable initialization deletion

Grundsätzlich können in Java die Instanzvariablen sowohl in der Variablendeklaration als auch im Konstruktor initialisiert werden. Um deren korrekte Initialisierung zu überprüfen, löscht der Mutationsoperator JID deren Instanzierung in der Variablendeklaration.[6]

Oiginal Code		**JID Mutant**
Class Stack {		Class Stack {
int size = 100 ;	Δ	int size;
Stack() {..}}		Stack() {..}}

Figure 4. JID[6]

JDC – Java-supported default constructor create

Entfernt implementierte Default-Konstruktoren, um deren korrekte Implementierung zu eruieren.[6]

4. Praktischer Einsatz (Tools)

Obwohl Mutationentests schon seit den 1970ern bekannt sind, werden sie außerhalb des akademischen Bereichs sehr selten verwendet. Dafür gibt es mehrere Gründe. Offutt und Untch identifizieren in [1] unter anderem folgende: das Fehlen wirtschaftlicher Anreize zur Verwendung „strenger“ Testmethoden, das Unvermögen Unit-Tests erfolgreich in den Softwareentwicklungsprozess einzubinden und Schwierigkeiten eine automatisierte Technologie zur Unterstützung von Mutationsanalyse und Mutationstesten zur Verfügung zu stellen.[1]

Wir wollen uns im Folgenden auf den dritten Grund konzentrieren: die automatische Unterstützung der Analyse und des Tests von Java-Programmen.

Ein Tool müsste grundsätzlich folgende Aufgaben ausführen können:

1. Den Sourcecode analysieren und die Mutanten erstellen.
2. Äquivalente Mutanten finden und eliminieren.
3. Die Testfälle mit dem Originalprogramm durchführen.
4. Wenn sie bestanden wurden, diese auch mit den Mutanten durchführen.[3]

Weiters muss festgehalten werden welche Mutanten erkannt werden konnten und welche nicht.[2]

Um den ersten Punkt, die Erstellung der Mutanten, zu erleichtern, wurde der „Metamutant“ entwickelt. Ein Metamutant ist ein Programm, dass alle Mutanten enthält. Über eine Umgebungsvariable wird bestimmt, welcher Mutant ausgeführt wird.[3] Weiters kann durch Konzepte wie „selective Mutation“ und „weak Mutation“ die Anzahl der Mutanten verringert werden.[1]

1 In [9] als AOC bezeichnet.

Wie im dritten Abschnitt der Arbeit gezeigt, wurden zur Erstellung der Mutanten Operatoren entwickelt, die häufige Fehler im Bereich der objektorientierten Programmierung, im Speziellen in Java, abdecken. Auch diese Voraussetzung für die Entwicklung eines Tools ist gegeben.

Eines der Hauptprobleme bei der Erstellung des Tools liegt im Erkennen und Auflösen von äquivalenten Mutanten. Durch automatische Methoden können bereits bis zu 60 % der äquivalenten Mutanten erkannt werden. Die restlichen Mutanten müssen noch von Hand identifiziert werden.[10]

Offutt, Kwnon und Ma haben vor kurzem ein Tool mit dem Namen µJava entwickelt. µJava erstellt aufgrund von 24, auf objektorientierte Fehler basierenden Operatoren objektorientierte Mutanten für Javaklassen. Nach der Erstellung der Mutanten können die Tests durchgeführt und die Testmenge überprüft werden. Die Mutanten werden automatisch erstellt und ausgeführt. Äquivalente Mutanten müssen jedoch per Hand identifiziert werden.[11] Aufgrund dieser Schwachstelle ist dieses Tool für den praktischen Einsatz wohl noch nicht interessant genug.

Man kann trotzdem behaupten, dass aufgrund der neuen Konzepte und der spezifischen Mutationsoperatoren ein Tool für Java, das sich für den praktischen Einsatz eignet, in greifbare Nähe gerückt ist. Es wird spannend sein zu beobachten, wie die Entwicklung auf diesem Gebiet voranschreitet.

5. Schlussfolgerung

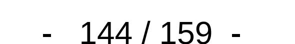
Zwei Faktoren werden unserer Meinung nach dazu beitragen, dass die Rolle von Mutationentests in Zukunft an Bedeutung gewinnen wird. Ein Faktor ist die Tatsache, dass Software vermehrt in Anwendungen, die hohe Verlässlichkeit fordern wie Life-Critical-Systems und E-Business-Systems, verwendet wird.[1] Der zweite Faktor sind die Fortschritte, die im Bereich des Mutationentests selbst erzielt werden konnten. Der nächste Schritt wird die Entwicklung eines Tools sein, das hohe Automation bietet und den Weg in die Praxis ebnen wird.

6. Glossar

Kopplungseffekt: Komplexe Fehler sind an einfache Fehler gekoppelt, sodass Testdaten, die alle einfachen Fehler in einem Programm entdecken, auch die meisten komplexen Fehler finden.[1]
Mutant: Fehlerhaftes Programm, das mithilfe eines Mutationsoperators aus dem Originalprogramm erzeugt wurde.[1]
Mutationsoperator: Eine Regel, die auf ein Programm angewandt wird, um einen Mutant zu erzeugen.[1]
selective Mutation: Es werden nur Mutanten erzeugt, die sich von anderen Mutanten wirklich unterscheiden.[1]
weak Mutation: Es wird nicht die Ausgabe des Mutanten mit der des Originalprogramms zum Schluss verglichen, sondern der interne Zustand der beiden sofort nach Ausführung des mutierten Bereichs im jeweiligen Programm.[1]

7. Referenzen

[1] A. J. Offutt, R. H. Untch, “Mutation 2000: Uniting the Orthogonal”, Symposium on Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries, San Jose, CA, October 2000, pp. 45-55.
[2] A. J. Offutt, “A Practical Tool System for Mutation Testing: Help for the common Programmer”, 12th International Conference on Testing Computer Software, Washington, D.C., June 1995, pp. 99-109.
[3] M. Bybro, “A Mutation Testing Tool for Java”, Examensarbete i Datalogi, Stockholm, August 2003.
[4] J. Offutt, S. D. Lee, “ An Empirical Evalutation of Weak Mutation”, In: IEEE transactions of Software Engineering, May 1994, 20(5): pp. 337-345.
[5] A. Snyder, “Encapsulation and Inheritance in Object-Oriented Programming Languages”, SIGPlan Notices, November 1986, pp. 38-45.
[6] Y. Ma, Y. Kwon, J. Offutt, “Inter-Class Mutation Operators for Java”, 13th International Symposium on Software Reliability Engineering, Annapolis, MD, November 2002.
[7] H. Gall, M. Hauswirth, R. Klösch, “Object-oriented concepts in Smalltalk, C++, Objective-C, Eiffel and Modula-3”, Technical University of Vienna, 1995.
[8] J. Gosling, et al., “The Java Language Specification - Second Edition”, 1996.
[9] S. Kim, J. Clark, J. McDermid, “Class Mutation: Mutation Testing for Object-Oriented Programs”, In: Object-Oriented Software Systems, Net.ObjectDays'2000, Germany, October 2000.
[10] J. Offutt, J. Pan, “Automatically Detecting Equivalent Mutants and Infeasible Paths”, In: The Journal of Software Testing, Verification, and Reliability, September 1997, Vol 7, No. 3, pp. 165-192.
[11] Y. Ma, Y. Kwon, J. Offutt, “MuJava : An Automated Class Mutation System”, 2004.



Reflexionen zum Bakkalaureatsseminar aus „Angewandte Informatik“ Sommersemester 2004

Roland Mittermeier
Institut für Informatik-Systeme
Universität Klagenfurt
roland@isys.uni-klu.ac.at

Abstract

Das erste Bakkalaureatsseminar aus “Angewandte Informatik” der Universität Klagenfurt wurde als Konferenzseminar organisiert. Dies erlaubte trotz hoher Zahl von Studierenden, die teils Bakkalaureanten, teils normale Seminarteilnehmer waren, allen eine angemessene Betreuung zuteil werden zu lassen. Diese Veranstaltungsform stellt aber auch spezifische Anforderungen an Studierende, was sich in einer hohen Zahl von Abbrüchen zeigte.

Dieser Beitrag erläutert die Form der Abwicklung und fasst jene Ergebnisse zusammen, die nicht unmittelbar aus den einzelnen Beiträgen der Studierenden erkennbar sind.

1. Einleitung

Das Universitätsstudiengesetz sieht vor, dass Bakkalaureatsarbeiten in Lehrveranstaltungen (Mehrzahl!) zu erbringen sind. Die Studienkommission Informatik der Universität Klagenfurt hat als solche

- das *Seminar aus Angewandter Informatik* und
- das *Praktikum aus Angewandter Informatik*

vorgesehen. Weiters führt der Studienplan aus, dass Bakkalaureatsarbeiten sich (je nach Lehrveranstaltungstyp) *in ihrem formalen Aufbau an einer wissenschaftlichen Publikation bzw. einem Projektbericht zu orientieren* haben.

Dieser Bericht bezieht sich auf das Seminar. In dieser Lehrveranstaltung ist daher laut Studienplan *eine theoretisch-konzeptionelle Arbeit, die ein Thema entsprechend dem Stand der Wissenschaft bzw. Technik aufarbeitet*, zu verfassen.

Da ich im Sommersemester des Jahres 2004 die Erstauflage eines solchen Bakkalaureats-Seminars leitete, es aber Prinzip ist, die Seminarleitung zwischen Informatikprofessoren rotieren zu lassen und darüber hinaus auch universitätsweites Interesse am Erfahrungsaustausch über die der neuen Rechtslage entsprechenden Form von Bakkalaureats-Seminaren besteht, möchte ich in diesem Aufsatz die dabei gewonnenen Erfahrungen der Kollegenschaft übermitteln. Es liegt an den nächsten Seminarleitungen, in wie weit das hier gewählte Modell aufgegriffen wird, ob es in modifizierter Form zum Einsatz kommt, oder ob völlig andersartige Durchführungsformen erprobt werden.

Da Konferenzseminare nicht zum seminaristischen Standardrepertoire gehören, möchte ich in der Folge diese Organisationsform allgemein beschreiben und anschließend auf die für dieses Seminar spezifische Variante eingehen. Dies bedeutet, dass nach Schilderung der Ausgangsbedingungen der konkrete Seminarablauf und die dabei gewonnenen Erfahrungen dargestellt werden. Ebenso werden die wesentlichsten Elemente des studentischen Feedbacks referiert.

2. Organisationsform

Die Idee, Seminare als Konferenzseminar durchzuführen, wurde von H. Pirker in einem im Rahmen der OOPSLA 1999 abgehaltenem Workshop über *educational patterns* aufgegriffen und nach Klagenfurt gebracht. Sie stieß auf Interesse der Forschungsgruppe *Software Engineering* und demgemäß fanden die Spezialseminare aus Software-Engineering in den Wintersemestern der letzten vier Studienjahre als Konferenzseminare statt.

Unter Konferenzseminar ist dabei zu verstehen, dass Studierende auf der Basis eines Rahmenthemas (Konferenzthema) in Beantwortung eines Call for Papers Beiträge einreichen und diese von einem Programmkomitee, dem auch Studierende angehören, begutachtet werden. Dieses Begutachtungsverfahren liefert jedenfalls breites Feedback. Das gilt sowohl für jene, deren Arbeiten akzeptiert werden, als auch für jene, deren Arbeiten abzulehnen sind. Aufgrund des Feedbacks aus dem Begutachtungsprozess können die Einreichungen bzw. die auf den schriftlichen Arbeiten fußenden Referate

überarbeitet werden. Diese Referate finden nicht, wie sonst in Seminaren üblich, laufend, im Wochenrhythmus, statt, sondern in einem Block am Ende der Veranstaltung.

Um eine allfällige Ablehnung frühzeitig zu erfahren, wurde bisher stets ein zweistufiges Begutachtungsverfahren eingesetzt. Nach Einreichung eines möglichst aussagekräftigen Extended Abstracts (siehe Seminarordnung, Anhang 1) war die Vollversion der Arbeit zu verfassen, die einer weiteren Begutachtung unterzogen wurde.

3. Ausgangsbedingungen

Die hier beschriebene Veranstaltung war ursprünglich als reines Bakkalaureatsseminar geplant. Es stellte sich jedoch heraus, dass die studentische Anfangspopulation, die dieses Seminar besuchen wollte, aus 23 Bakkalaureatsstudierenden der Informatik, 28 Diplomstudierenden, 3 Lehramtskandidatinnen und 14 Betriebswirtschaftsstudierenden bestand. Da dies weit über die für ein Seminar zuträgliche Zahl von Studierenden hinausging und auch qualitativ aufgrund der unterschiedlichen Vorkenntnisse und unterschiedlichen Lehrziele eine untragbare Situation ergeben hätte, wurde für die Betriebswirtschaftsstudierenden ad hoc ein eigenes, von dieser Veranstaltung getrenntes Seminar angeboten. Die verbliebenen Studierenden hatten zwar weiterhin unterschiedliche Einstiegsvoraussetzungen, doch immerhin ein im Wesentlichen gemeinsames Ausbildungsziel. Den Diplomstudierenden wurde angeboten, in Spezialseminare aus den gebundenen Wahlfächern zu wechseln. Doch von dieser Option machten nur wenige Gebrauch, sodass die Veranstaltung letztlich mit 38 Studierenden (18 B + 20 S) den Lehrbetrieb aufnahm.

Das Rahmenthema „Software-Qualität“ wurde bereits frühzeitig bestimmt. Ebenso wurde die Abhaltung als Konferenzseminar schon innerhalb der Sommerferien festgelegt, als sich abzeichnete, dass das Seminar mit einer Mischpopulation von Studierenden abzuhalten ist.

Im Gegensatz zu den bisher als Konferenzseminar abgehaltenen Spezialseminaren aus Software-Engineering, bei denen außer der Vorbesprechung, der (extramuralen, geblockten) Präsenzphase und einer unmittelbar davor stattfindenden Organisationsbesprechung die gesamte Betreuung der Teilnehmer über e-mail, BSCW und im WS 2003/04 auch über ein Konferenzmanagementsystem abgewickelt wurden, wurden hier im wesentlichen wöchentliche Präsenztermine vorgesehen. Der Grund für die unterschiedliche Durchführungsart lag einerseits in der hohen Teilnehmerzahl, bei der eine rein elektronische Betreuung zu aufwändig wäre (Individualkommunikation via e-mail). Andererseits schien bei den im Studium jüngeren Bakkalaureatsstudierenden und auch aufgrund des Faktums, dass es sich um eine Pflichtlehrveranstaltung handelte, eine rein elektronische Abwicklung als zu riskanter Schritt.

4. Ablauf

Die Seminarordnung, die diese Veranstaltungsform festlegte und der Call for Papers (siehe Anhang 2) wurden bereits vor Lehrveranstaltungsbeginn über das Content-Managementsystem CLAROLINE für die Zielgruppe publiziert und in der ersten Veranstaltung erläutert.

Hiezu ist zu bemerken, dass das Programm nur bis zu Beginn der Osterferien genau spezifiziert wurde, da bei der zu Lehrveranstaltungsbeginn vorliegenden Teilnehmerzahl die Abhaltung der Vortragsphase unrealistisch hohen Zeitaufwand erfordert hätte, die Lehrveranstaltungsleitung allerdings davon ausgehen konnte, dass sich die Teilnehmerzahl im Zuge des Begutachtungsprozesses wohl reduzieren würde. Das Ausmaß dieser Reduktion konnte freilich nicht antizipiert werden. In ihrem tatsächlich eingetretenen Umfang war dies sehr bedauerlich. Auch war vorab nicht klar, wie viel Studierende das Angebot, in ein anderes für Diplomstudenten angebotenes Seminar zu wechseln, annehmen würden.

Während der beiden ersten Wochen wurden Themenvorschläge der Studierenden erfragt und vom Lehrveranstaltungsleiter kommentiert. Neben dem Ziel, bei der Findung eines bearbeitbaren Themas behilflich zu sein (zu weit, zu eng, Titel zu unklar oder zu wenig attraktiv) und allenfalls generelle Hinweise zur Literatursuche bezüglich des gewählten Themas zu geben, hatte das wöchentliche Treffen in dieser Phase vor allem das Ziel, sicherzustellen, dass die Studierenden diese Vorbereitungszeit aktiv nutzen. Eine Vorsichtsmaßnahme, die sich nicht für alle, wohl aber – wie die Art einiger Themennennungen in der zweiten Seminarwoche vermuten ließen – doch für einige Studierende als notwendig erwies.

Im studentischen Feedback (teiloffener Fragebogen) mag einer dieser beiden Veranstaltungstage als „überflüssiger Präsenztermin“ kritisiert worden sein. Ich würde ihn dennoch nicht entfallen lassen. Sehr bedenkenswert erscheint jedoch die Kritik, dass die dafür aufgewendete Zeit dazu führte, dass ein Referat über methodische Hinweise erst nach Einreichung der Extended Abstracts gegeben werden konnte. Ein solches Kurzreferat ist unbesehen dessen, dass alle Studierenden bereits mindestens ein Proseminar aus Informatik absolviert hatten, erforderlich. Allerdings wurden Hinweise zur Textsorte „Extended Abstract“ auch in den bisherigen Spezialseminaren, nicht zuletzt aus Gründen der ex post höheren Aufmerksamkeit, erst nach der ersten Feedback-Runde gegeben, dort allerdings elektronisch in Schriftform.

Der Begutachungsprozess der Extended Abstracts verlief aus Sicht der Lehrveranstaltungsleitung unspekta-

kulär. Die Ergebnisse variierten zwischen den Teilnehmern jedoch so deutlich, dass die Ankündigung, die Qualität der Gutachten in die Seminarnote einfließen zu lassen (siehe Seminarordnung) jedenfalls als sinnvoll und notwendig anzusehen ist. Um auch diesbezüglich für Klarheit zu sorgen, wurde in der Woche nach den Osterferien (die Gutachten für die Extended Abstracts waren bereits zugestellt) ein Punktezwischenstand bekannt gemacht. Dies bot auch jenen (wenigen), die sehr divergente Gutachten erhielten, die Chance, zu sehen, wie die Lehrveranstaltungsleitung, die ja nur ein normales Gutachten abgab, ihre Einreichung beurteilte. Allerdings ist darauf hinzuweisen, dass die Mehrzahl der studentischen Gutachten hohe, einige sogar extrem hohe Qualität hatten.

Festzuhalten ist, dass jede Arbeit von mindestens drei studentischen Reviewern und vom Lehrveranstaltungsleiter begutachtet wurde. Um die Belastung der studentischen Gutachterinnen und Gutachter mit jeweils drei Arbeiten konstant zu halten bekamen allerdings Arbeiten, die von Paaren eingereicht wurden, eine übertrieben hohe Anzahl von Gutachten.

Nach den Osterferien kam es zu einem drastischen Schwund an Teilnehmern. Dieser wurde teils unterschiedlich begründet, teils erfolgte er stillschweigend. Der Hinweis, dass man mit Plagiatrie keinesfalls zu einem positiven Abschluss gelangen kann (siehe Abschnitt Erfahrungen, freie Themenwahl) mag dazu einiges beigetragen haben. Jedenfalls waren zwei Wochen nach Wiederaufnahme des Lehrbetriebs nur mehr 8 Bakkalaureatsstudierende und 18 Seminaristen (7 Zweier-Teams und 4 Einzelvortragende) verblieben. Diese Personen beendeten das Seminar auch (eine Zweiergruppe jedoch negativ).

Um den laufenden Arbeitsfortschritt sicherzustellen und allenfalls weiteres inhaltliches Feedback, insbesondere auf Reaktionen auf den Begutachtungsprozess zu geben, wurden für die Phase nach den Osterferien Kurzpräsentationen anberaumt. Für diese standen pro Seminarthema strikt maximal 4 Minuten (pro Bakkalaureatsthema 5 min) bzw. 3 Folien, die vorab abzugeben waren um rasche Präsentationsfolge zu gewährleisten, zur Verfügung. Der in einer e-mail gemeinsam mit einer Pauschalkommentierung der Begutachtungen in der Karwoche angekündigte Kurzvortragsblock verfolgte neben den erwähnten Zielen auch die Absicht, die für die Präsentationen unbedingt erforderliche Einhaltung von Vortragseffizienz sowie das Einhalten der zeitlichen Rahmenbedingungen zu sichern. Inhaltliches wurde im Anschluss an diese Referate nur in wenigen Fällen diskutiert. Weit eher konnten unterschiedliche Präsentationsstrategien (Lösungen der Folien und Zeitbeschränkung) analysiert und besprochen werden.

Im Anschluss an die Kurzpräsentationen wurde der Zeitplan der Vortragsphase festgelegt und (abgesehen von einem „Fragetermin") bis zu dieser keine weitere Präsenzveranstaltung. Der „Fragetermin" hätte dabei wohl entfallen können. Ein inhaltlich nur bedingt ausgefüllter Termin unmittelbar nach Begutachtung der Extended Abstracts ist jedoch sinnvoll.

Die Einreichung und der Begutachtungsprozess der Vollbeiträge erfolgte rein elektronisch (wie bei Extended Abstracts). Die Varianz der Gutachten war weit geringer als in der ersten Gutachtensrunde bei durchschnittlich hoher Qualität.

Die Qualität der Einreichungen war hingegen sehr unterschiedlich. Insbesondere von den Bakkalaureatsarbeiten waren noch zu viele als zu problematisch zu beurteilen, als dass sie als akzeptabel geschweige denn als sehr gut zu beurteilen gewesen wären. Es wurde daher allen eine Nachfrist eingeräumt, die Arbeit bis 1. Juli zu überarbeiten. Von dieser Option haben alle Bakkalaureanten (auch jene mit sehr guten Ausgangsarbeiten) und zwölf Seminaristen Gebrauch gemacht.

Die Präsentationsphase (Programm im Anhang) verlief sehr diszipliniert, allerdings viel zu gedrängt. Dies hatte zur Folge, dass – im Gegensatz zu bisherigen Konferenzseminaren – inhaltlich nur relativ wenig zu den einzelnen Vorträgen diskutiert wurde. Der Zeitplan erlaubte kaum mehr als eine inhaltliche Frage und kurze Statements zur Vortragstechnik, sodass dieses Defizit nicht zu Lasten der Studierenden ausgelegt werden kann.

5. Erfahrungen

In diesem Teil möchte ich Erfahrungen genereller Art für jene, die diese Veranstaltungsform übernehmen und weiterentwickeln wollen, stichpunktartig zusammenfassen:

- *Freie Themenwahl*: Sie wurde im Rahmen des Feedbacks von den Studierenden mehrfach als Positivum erwähnt. Aus Sicht des Lehrveranstaltungsleiters muss jedoch ergänzt werden, dass damit auch Risken verbunden sind. Einige Extended Abstracts standen unter begründetem Verdacht, dass die geplante Arbeit sehr plagiatsverdächtig sein würde. Eine Person wurde sogar von der weiteren Teilnahme ausgeschlossen, weil bereits das Extended Abstract eine Plagiats-Collage war.
  Wie weit diese seminaröffentliche Plagiatsbehandlung dazu führte, dass einige Studierende nach der Reviewphase der Extended Abstracts das Seminar verließen und wie weit Kritik am jeweiligen Extended Abstract oder der offenkundige Zwang, kontinuierlich an der Lehrveranstaltung zu arbeiten dazu führte, kann (und soll) nicht weiter analysiert werden.
- *Striktes Terminmanagement*: Dieses war zwar vielleicht für einige Studierende unüblich. Es wurde je-

doch, sei es der Sache wegen, sei es aus Einsicht über die zu betreuende Studierendenzahl, akzeptiert (sogar punktuell gelobt).

- *Vertraulichkeit*: Der Gutachtensprozess war als blinder Review organisiert und die Studierenden wurden darauf aufmerksam gemacht, dass sie bei besonderen Naheverhältnissen um Tausch der Zuweisung ersuchen sollten. Verletzungen dieser Spielregeln wurden nicht augenscheinlich. In einigen Fällen lassen Indizien auf das strikte Einhalten der Vertraulichkeit schließen.
  Dies bedeutet freilich, dass das Begutachtungsverfahren seitens der Lehrveranstaltungsleitung auch tatsächlich mit Ernst und Sorgfalt eines Begutachtungsverfahrens für eine echte Konferenz durchzuführen ist.
- *Review-Formulare*: Es wurden „echte“ Begutachtungsformulare verwendet (nur minimale Adaptionen). Dies erhöht die Glaubwürdigkeit und war bei Spezialseminaren durchaus adäquat. Für zu viele hier eingereichte Übersichtsarbeiten war dies jedoch nicht ganz passend.
  Ich würde daher wenigstens für die Begutachtung der Extended Abstracts im Wiederholungsfall speziell auf diesen Lehrveranstaltungstyp zugeschnittene Review-Formulare einsetzen (für die Endausarbeitung scheinen „echte“ Formulare durchaus passend).
- *Format und Seitenvorgaben*: Beides hat sich m.E. bewährt. Das Limit mit 6 Seiten für Zweierteams und 10 Seiten für Bakkalaureatsarbeiten würde ich beibehalten. Jenes von 4 Seiten für Einzelvortragende scheint zwar fair zu sein, ist aus meiner Sicht jedoch etwas problematisch. Auch höhersemestrigen Studierenden fehlt offenbar noch die Routine, auf so engem Raum ein Thema in gebotener Tiefe darzustellen. Es bleibt jedoch den Leserinnen und Lesern dieses Readers überlassen sich darüber ein eigenes Urteil zu bilden.
- *Zeitplan*: Der Zeitplan war durch die Überbelegung zu Beginn der Veranstaltung einerseits, durch die Zäsur der Osterferien andererseits geprägt. Dies ergab einerseits, dass die Zeit für die Begutachtung der Langfassungen sehr knapp war (etwas mehr als eine Woche wäre wünschenswert) und dass für die Präsentationsblöcke zu dem Zeitpunkt, als sie geplant werden konnten, keine Termine für Tages- oder Halbtagsblöcke mehr zur Verfügung standen. Es wäre jedenfalls wünschenswert, hier früher planen zu können und daher rechtzeitig (also noch vor anderen Blocklehrveranstaltungen) etwa „Präsentations-Samstage“ zu definieren.
- *Präsenzphase*: In den bisherigen Spezialseminaren war die Präsenzphase ein extramuraler Block, in dem meist mehr als die Vortragszeit auf nachfolgende inhaltliche Diskussionen verwendet wurde. Schließlich ist durch den Begutachtungsprozess ja bei einem Teil des Auditoriums bereits ein hoher Bezug zum Referat vorhanden und diese reißen die anderen mit.

  Wie erwähnt war dieser Effekt bei der hier gewählten Form der zeitlich verteilten Präsentation im Seminarraum nicht gegeben. Die Begründung scheint jedoch im Zeitdruck gelegen zu sein, der aufgrund der hohen Teilnehmerzahl entstand. Wegen des erwarteten Teilnehmerschwunds konnte in diesem Fall nicht rechtzeitig ein Präsentationsblock geplant werden, sodass lediglich die Option des frühen LV-Beginns übrig blieb.
  Anzumerken ist noch, dass die Vortragsreihenfolge nicht ident zur hier gewählten Präsentationsreihenfolge war. Das geschlossene Vorziehen der reinen Seminaristen ergab sich rein daraus, dass diese doch weniger umfangreiche Arbeiten auszuarbeiten hatten. Hier wurde auch noch in Ansätzen eine thematische Gruppierung vorgenommen. Bei den Bakkalaureatsarbeiten wurde die Vortragsreihenfolge eher nach Kriterien des Stands der Erstausarbeitungen getroffen, sodass an den jeweiligen Tagen sowohl Studierende mit guten Zwischenleistungen als auch Studierende mit Verbesserungsbedarf der Erstfassung der Bakkalaureatsarbeit präsentierten. In Abwägung der Qualitätsverbesserungen der meisten zur zweiten Kategorie gehörenden Arbeiten scheint dieses Konzept aufgegangen zu sein.

6. Studentisches Feedback

Gegen Ende der Veranstaltung wurde den Studierenden ein Fragebogen zur Verfügung gestellt, der in geschlossenen Fragen die Relation von Ertrag zu Aufwand dieser Veranstaltung bzw. Veranstaltungsform erhob und darüber hinaus drei Felder für Freitext zur Frage „was hat mir gefallen“ und „was hat mir nicht gefallen“ erhob.

Die Befragung wurde von 7 von 8 Bakkalaureatsstudierenden und von 8 von 16 Seminaristen beantwortet. Eine Person füllte die Charakterisierung Bakkalaureat oder normales Seminar nicht aus. Dies bedeutet, 10 Studierende, also etwa ein Drittel der Studierenden beteiligten sich an dieser Befragung nicht.

Im Durchschnitt hatten die Befragten vor dieser Veranstaltung 3,9 Proseminare und 1,8 Seminare besucht. Dabei war bei den Bakkalaureatsstudierenden die Proseminar-Erfahrung mit 4,6 deutlich höher als bei den Seminaristen (3,4). Seminaristen hatten jedoch im Vorfeld im Schnitt bereits 2,4 Seminare besucht, während Bakkalaureatsstudierende lediglich 1,4 Seminare vorab

absolvierten[1]. Damit wurde die Vermutung der Lehrveranstaltungsleitung, dass Seminaristen bereits über höhere Seminarerfahrung verfügten als die Bakkalaureatsstudierenden und daher insbesondere in der Anfangsphase der Lehrveranstaltung weniger leichter Tritt fassten, ein wenig bestätigt. Dieser Befund sollte wohl in künftigen Bakkalaureatsseminaren besondere Berücksichtigung finden.

Die Antworten bezüglich Aufwand und relativer Ertrag der Lehrveranstaltung sind in der nachfolgenden Tabelle dargestellt. Hiezu ist zu bemerken, dass die Gesamtzahl der Antworten stets um 1 größer ist, als die Summe der Antworten der Bakkalaureats- und Seminar-Studierenden, da eine Person dieses Zuordnungsfeld nicht ausfüllte und daher nur in der Gesamtsumme aufscheint.

stud. Aufwand vs. Ertrag	**Bakk. Stud.**	**Seminaristen**	**gesamt**
Ertrag relativ zu anderen seminaristischen LVs			
etwa gleich	2	2	4
mehr	5	6	12
Ertrag relativ zu durchschnittlicher LV			
weniger	1	-	1
etwa gleich	4	6	10
mehr	1	2	4
viel mehr	1	-	1
Aufwand relativ zu anderen seminaristischen LVs			
gleich	1	3	4
höher	4	4	9
viel höher	2	1	3
Aufwand relativ zu durchschnittlicher LV			
weniger	-	1	1
gleich	1	3,5	4,5
höher	4	3,5	8,5
viel höher	2	-	2
Nutzen			
gelohnt	4	5	10
annähernd o.k.	3	2	5
nicht gelohnt	-	-	-
no response	-	1	1

Dabei wurden die ersten vier Fragen auf einer fünfteiligen Skala (*viel weniger, weniger, etwa gleich, mehr, viel mehr),* der Nutzen auf einer dreiteiligen *(sicherlich gelohnt, war annähernd o.k., nicht gelohnt)* abgefragt.

Unter den frei formulierten Antworten zu ***gefallen hat mir***, stechen 9 Nennungen der Art *viel Feedback zu den Arbeiten* hervor. Diese sind noch durch 2 Nennungen von *Reviews*, 2 Nennungen zu *Analyse der Präsentation* und 3 Nennungen *Kommentare des LV-Leiters* zu ergänzen. (Letzteres wird allerdings durch 2 Nennungen des LV-Leiter Feedbacks unter *nicht gefallen* relativiert). Eine wietere reviewbezogene Antwort lautete *Review anderer Arbeiten*. Damit kann der gekoppelte Begutachtungsprozess durch Kommilitonen wie Lehrveranstaltungsleitung als deutlich positiv beurteilt angesehen werden.

Weitere Mehrfachnennungen betrafen die Veranstaltungsform einer *Konferenz* (dreimal) und die *freie Themenwahl* (zweimal am Fragebogen genannt, in der Abschlussdiskussion jedoch deutlich unterstrichen). Auch Antworten wie *regelmäßige Abgaben und regelmäßiges Feedback* oder *Pflicht, Zeit exakt einzuhalten* und zweimaliges Nennen von *klare Vorgaben* gehören in diese Kategorie. Ebenso fallen in diese Kategorie wohl zwei Nennungen bezüglich des *Umfangs der Arbeit,* die die Längenbeschränkung als sinnvoll bezeichnen.

Drei Antworten bemerkten positiv, *gelernt zu haben, mit Papers informatischen Inhalts zu arbeiten* (verfassen, auffinden, ...). Die restlichen postitiven Antworten betrafen das *Arbeitsklima* und Aspekte, die wohl in jedem Seminar gegeben sein sollten.

Die Antworten zu ***NICHT gefallen hat mir*** waren breiter gestreut, hatten jedoch mit 6 Nennungen von *früher Beginnzeitpunkt* einen deutlich herausragenden Modus. Dieser ist gefolgt von drei Nennungen von *zu viele Präsenztermine* und zwei Nennungen von *geheimer Reviewprozess* mit Zusatz *namentlich wäre objektiver* bzw. *besser, wenn jeder weis, von wem Reviews stammen*. Eine Person meinte, die Zeit für die Begutachtung der Langfassungen wäre zu knapp bemessen gewesen. In diese Kategorie gehört wohl auch *zu viele Reviews pro Person (lieber 2 mit mehr Zeit)*. Die Antwort *teilweise unklare Aufgabenstellungen/Fragen bei Reviews* geht wohl auf die Verwendung von Originalformularen zurück, die, wie erwähnt, an einigen Stellen nicht ganz für die studentischer Themenbearbeitungen passten.

Einen anderen Komplex berühren Antworten wie *Mischung Bakkalaureatsseminar mit Normalseminar* (zwei Nennungen) oder großer *Aufwandsunterschied zwischen 2h und Bakk Seminarteilnehmern in Relation zum Stundenumfang*. In diese Kategorie gehört wohl auch die unter ***darüber hinaus möchte ich zu dieser Lehrveranstaltung noch festhalten*** getroffene Kritik *zu harte Beurteilung vor allem der Bakkalaureatsstudierenden*.

Dort findet sich auch die Bemerkung, dass *bei Kurzpräsentationen nicht klar war, was präsentiert werden*

[1]) Im Bakkalaureatsstudium Informatik ist dieses Bakkalaureatsseminar das einzige Pflichtseminar. Die Vorerfahrung stammt daher aus anderen Fachgebieten (Anwendungsfach) oder aus Studienplänen, nach denen vor einem allfälligen Wechsel in das Bakkalaureatsstudium studiert wurde.

soll. Dies ist richtig, gab es doch lediglich formale Vorschriften (Zeitdauer, Folienanzahl). Doch gerade das dadurch entstandene Spektrum bot Diskussionsstoff und zeigte auch, dass durchaus unterschiedliche Strategien sehr positive Effekte erzielen können.

Auch die Bemerkung *Ziel bzw. Thema musste selbst definiert werden, daher lang auf dem Holzweg* mag für mehr als nur eine Person gegolten haben, wurde jedoch nur von einer negativ vermerkt. Bei all den anderen dominierte offenbar die Motivation, sich durch ein selbst gewähltes Thema durchzubeißen.

Insgesamt, und dies ergab wohl auch die abschließende Diskussion mit den Studierenden, wurde die Veranstaltung trotz einer für eine reale Konferenz erforderlichen rigiden Führung mehrheitlich positiv beurteilt.

6. Zusammenfassung

Insgesamt darf festgestellt werden, dass die Form der Abhaltung des Bakkalaureatsseminars als Konferenzseminar erfolgreich war. Auch wenn in der Abschlussdiskussion die Meinung vertreten wurde, die Abfassung einer Bakkalaureatsarbeit im Stile einer Diplomarbeit hätte Vorteile, scheint doch für die Mehrheit der Studierenden das in dieser Veranstaltungsform gegebene studentische Feedback eine wichtige Orientierungshilfe zu sein.

Die relativ freie Themenwahl ist motivierend, hat jedoch auch ihre Gefahren, auf die Lehrveranstaltungsleitungen vorbereitet sein sollten. Man mag fragen, ob man die damit verbundenen Risken in einer Pflichtlehrveranstaltung eingehen möchte. Das Argument, dass dies für jene Studierenden die mit dem Bakkalaureat abschließen, die letzte (?einzige?) Chance im Studium ist, in Eigenverantwortung eine kleine wissenschaftliche Arbeit zu versuchen, sollte dabei berücksichtigt werden. [2]

Bei etwas geringerer aber stabilerer Teilnehmerzahl sollte es auch möglich sein, den in als Konferenzseminar organisierten Spezialseminaren erzielten intensiven Diskurs zu den Präsentationen in das Bakkalaureatsseminar zu übertragen, sodass auch dieser Aspekt der Vorinformation durch das Begutachtungsverfahrens genützt werden kann.

[2]) Interessant mag unter diesem Aspekt ein Vergleich von § 61a UniStG 99 mit seiner Referenz auf § 61 (2), freie Wahl des Diplomarbeitsthemas, mit § 80 UG 2002, sein. Dieser verweist nicht mehr auf § 81 (Diplomarbeiten). Dieser deutet jedoch in Abs. 2 eine Festlegung des Themas durch den Betreuer an oder lässt allenfalls offen wer die Aufgabenstellung so wählt, dass die Bearbeitungsfrist eingehalten werden kann. Es wird wohl am Selbstverständnis der Universitäten und ihrer Angehörigen liegen, wo die Grenze zwischen Freiheit und Lenkung künftig gezogen wird. Zu hoffen bleibt, dass nicht quantitative Restriktionen dazu führen, dass von der gesetzlichen Möglichkeit der Engführung zu restriktiv Gebrauch gemacht werden muss.

Anhang 1

Seminar aus Angewandter Informatik
Seminar aus Betriebsinformatik
R. Mittermeir
Sommersemester 2004

Beginn
Di, 2. März 2004; 8:15
Seminarraum 2.42

Seminarordnung

Das Seminar wendet sich an vier Zielgruppen mit durchaus unterschiedlichen Vorkenntnissen und auch unterschiedlichen Zielvorstellungen.

Primär wollte es die Studienkommission als

> **dreistündige** Lehrveranstaltungen für Bakkalaureats-Studierende aus Informatik

konzipiert wissen. Darüber hinaus ergab sich der Bedarf es als

> **zweistündige** Lehrveranstaltung für Lehramtsstudierende, Betriebswirtschafts-Studierende und ggf. auch für Diplomstudierende aus Informatik, die noch im zehnsemestrigen Diplomstudium sind und nicht die Ausweich-Variante eines zusätzlichen Seminars aus dem gebundenen Wahlfach annehmen wollen bzw. können

zu gestalten.

Der Unterschied im Stundenausmaß ist dabei insofern das geringere Problem, als aufgrund des Bakkalaureats-Studienplans die Aufstockung des Stundenausmaßes um eine Stunde mit der Notwendigkeit der Abfassung einer über die Anforderungen an eine Seminararbeit hinausgehenden Bakkalaureatsarbeit begründet ist.

Um sowohl den Anforderungen der Bakkalaureatsarbeit schreibendenden als auch die Bedürfnisse und Kenntnisse der anderen unterschiedlichen Zielgruppen fair zu berücksichtigen, wird das Seminar als **Konferenzseminar** angeboten. Dies bedeutet: Ausgehend von einem für alle Gruppen (hoffentlich) interessanten und (jedenfalls) bewältigbaren Themenschwerpunkt (Software-Qualität; siehe Aufforderung zur Beitragseinreichung)

- bestimmen Sie ihr konkretes Seminar-Thema,
- werden Sie von der Seminarleitung gemeinsam mit den Kommilitonen betreut
- es liegt (vielleicht etwas stärker als sonst üblich) in Ihrer Hand, wie sich das Seminar entwickelt.

Konkret bedeutet dies folgendes **Zeit- und Arbeits-Raster**:

2. März. 04 — Vorbesprechung und Gruppeneinteilung für all jene, die ein zweistündiges Seminar besuchen wollen. (Bakkalaureanten und jene, die sich die Lehrveranstaltung als 3-stündiges Seminar anrechnen lassen wollen, arbeiten als Einzelpersonen).

9. März 04 — Methodische Hinweise, Besprechung von Einzelthemen der Referate

16. März 04 Fixierung der Einzelthemen (Arbeitstitel der Referate)
22. März 04, 8:30 Uhr extended abstracts sind eingereicht **(Fallfrist!!!)**
23. März 04 Hinweise zum Reviewing von Abstracts
30. März 04 Kurzpräsentationen, Fragestunde
1. April 04, 8:30 UhrReviews der extended abstracts fällig **(Fallfrist!!!)**
7. April 04 Feedback zu extended abstracts bei (Erst-)Autoren
Osterferien
Ausarbeitung der Langfassungen
Dienstage nach Vereinbarung (1. April) Zwischen- und Kurzpräsentationen mit Feedback aus Plenum (5 min Vorträge)
ab Mitte Mai: Fälligkeitstermine für Langfassungen (unterschiedliche Termine für unterschiedliche Zielgruppen)
10 Tage Reviewing-Periode
4 Tage Zusammenfassungs-Periode
ab Anfang Juni (voraussichtlich mit Zusatz-Terminen): Vortrags-Termine

Abzugebendes Material

Extended Abstracts: Titel, halb- bis max. einseitige Darstellung über Ziel und Inhalt der Arbeit (Orientierungsgröße: 2000 bis 3000 Zeichen), Inhaltsverzeichnis (zweistufig), Angabe der drei wichtigsten Arbeiten, auf die sich das Referat stützen wird.

Reviews der extended abstracts: Auf bereitgestelltem Formular

Langfassung: Für Kurzbeiträge (zweistündiges Seminar) maximal vier Seiten, für Vollbeiträge (Bakkalaureatsarbeiten, dreistündiges Seminar) acht bis zehn Seiten in klassischem acm oder IEEE Proceedings-Style (wird bereitgestellt). Das selbst gesteckte Ziel und der dafür erforderliche Inhalt ist innerhalb dieses Platzes (samt erforderlicher graphischer Darstellungen) unterzubringen.

Reviews der Langfassungen

Vortrag mit zugehörigem Präsentationsmaterial: Vortragsdauer für Kurzbeiträge max. 15 min, für Hauptvorträge (Bakkalaureats-Vorträge) max. 30 min.

Beurteilungsschema
Beurteilt wird die auf Individualleistung und Mitarbeit beruhende Gesamtleistung bestehend aus

- Qualität des „extended Abstracts“ 20 %
- Leistungen im Reviewprozess
  . Begutachtung d. extended Abstracts und
  . Begutachtung d. full papers 20 %
- Full paper
  . submitted version + final version 30 %
- Vortrag 20 %

- laufende Mitarbeit, Diskussion 10 %

wobei allerdings zu berücksichtigen ist, **dass eine positive Seminarteilnahme nur** dann gegeben ist, **wenn in allen genannten Beurteilungsdimensionen** die erforderlichen Beiträge **fristgerecht** erbracht wurden.

Allfällige offene Punkte können in der Vorbesprechung geklärt werden.

***Anmerkung**: Der Zeitplan wurde nach den Osterferien für den Rest des Monats April und für Mai wie folgt festgelegt. Zu diesem Zeitpunkt war zwar die Zahl der Normalseminaristen bereits stabil, der Dropout an Bakkalaureatsstudierenden jedoch noch nicht manifest. Daher wurden die Vortragsblöcke (Juni) vorerst noch nicht genauer definiert.*

20. April 04 Kurzpräsentationen – Diplomstudierende (4 min)
27. April 04 Kurzpräsentationen – Bakkalaureatsstudierende (5 min)
4. Mai 04 offen (wird am 27. Apr. besprochen)
10. Mai 04, 8:30 Uhr Langfassungen der Diplomstudierenden sind eingereicht **(Fallfrist!!!)**
11. Mai 04 Begutachtungsaspekte, Vortragsprogramm
14. Mai 04, 13:00 Uhr Reviews für „2h-Arbeiten“ sind fällig (**von allen!**) **(Fallfrist!)**
17. Mai 04, 8:30 Uhr Langfassungen der Bakkalaureatsarbeiten sind eingereicht **(Fallfrist!!!)**
17. Mai 04, (später Nachmittag, Abend) Feedback für „2h-Arbeiten“
18. Mai 04 offen (wird am 11. Mai besprochen)
24. Mai 04, 8:30 Uhr Reviews für „Bakkalaureatsarbeiten“ sind fällig (von allen!) **(Fallfrist!)**
25. Mai 04 Vorträge – „2h-Arbeiten“ – 1. Runde (Vortragsdauer: 15 (max. 20) min)
weiteres Programm, soweit bis dahin nicht bereits festgelegt,
wird am 25. Mai besprochen.
Bakkalaureats-Vorträge beginnen ab 8. Juni 04 (25 bis max. 30 min)

Anhang 2

Seminar aus Angewandter Informatik
Seminar aus Betriebsinformatik
R. Mittermeir
Sommersemester 2004

Beginn
Di, 2. März 2004; 8:15
Seminarraum 2.42

Software-Qualität

Call for Contributions / Aufruf zur Beitragseinreichung

Für das Seminar aus Angewandter Informatik / Betriebsinformatik werden Beiträge aus dem Bereich Software-Qualität erbeten.

Bedenken Sie dabei, dass Software Qualität aus unterschiedlichen Aspekten beleuchtet werden kann. Entsprechend der sehr unterschiedlichen Teilnehmer-Population am Seminar erwarte ich Beiträge zu

- anwendungsbezogenen Themen, wie etwa
  - Qualität im Sinne von Usability
  - Qualität im Sinne von Integrierbarkeit in die betriebliche Umwelt
  - Qualität im Sinne von Prozessverbesserung innerbetrieblicher Abläufe durch Software
  - Qualität und Software-Release-Politik
- prozessbezogenen Themen, wie etwa
  - Qualität des Software-Entwicklungsprozesses
  - Zertifizierung des Software-Entwicklungsprozesses (CMM, ISO, SPICE, …)
  - Qualität einzelner Phasen des Prozesses
  - Life-Cycle übergreifendes Qualitätsmanagement
- techniken-spezifische Themen, wie etwa
  - Requirements Elicitation
  - Requirements Feedback
  - Requirements Tracing
  - spezifische Review-Techniken
  - spezifische Test-Verfahren
  - Aspekte der Software-Integration im Zusammenhang mit CBSE
  - Software Metriken (und ihr Bezug zu Qualität)
- sonstige Themen mit Fokussierung auf Software-Qualität

Bitte beachten Sie, dass, gleichviel für welche Themengruppe sie sich entscheiden und unabhängig von Ihrer Zuordnung zu einer der vier studentischen Kategorien eine klar fokussierte Arbeit erwartet wird. Stecken Sie sich daher kein zu breites Thema. Wählen Sie vielmehr einen Bereich, in dem Sie sich vertiefen wollen und zu dem Sie Aussagen aus der Literatur reflektierend kommentieren oder an eigenen Erfahrungen messen.

Seminararbeiten (Bakkalaureatsarbeiten erst recht!) sind kleine eigenständige wissenschaftliche Arbeiten, die in der jeweils relevanten Fachliteratur wurzeln und in eigenständigen Beiträgen

gipfeln. Damit solche Gipfel auch erreichbar sind, dürfen sie nicht zu breit angelegt werden. Anderseits muss die Basis natürlich breit genug und ausreichend tragfähig sein. Keinesfalls sind Seminar-/Bakkalaureatsarbeiten nett vorgetragene Nacherzählungen!

Lehrziel ist, dass Studierende durch selbständige Bearbeitung eines aktuellen Themas aus Angewandter-/Betriebs-Informatik neben fachlichen Kenntnissen auch methodische Kenntnisse im Verfassen wissenschaftlicher Arbeiten (etwa der Diplomarbeit) erwerben, bzw. auf Grundlage und im Rahmen des Seminars ihre Bakkalaureatsarbeit (theoretischer Teil) verfassen. Ebenso soll durch positive Kritik solcher Arbeiten bereits während der Entstehungsphase Führungs- und Teamfähigkeit geübt werden, aber auch unmittelbar Beiträge zum Erfolg der Gesamtveranstaltung geliefert werden.

Methodisch baut dieses Seminar auf den Erfahrungen der Abwicklung von Konferenzseminaren (Organisationsform des Seminars aus Software Engineering der letzten Jahre) auf. Aufgrund der unterschiedlichen Zielgruppen und auch aufgrund der hohen Teilnehmerzahl ist die Abwicklung während des Semesters nicht rein elektronisch. Dennoch wird es eine umfangreiche Vorbereitungsphase geben, auf die einige Referate-Blöcke aufbauen.

Details zur Abwicklung, Terminplan und Beurteilungsraster finden Sie in der Seminarordnung (in CLAROLINE). Auf die Unabdingbarkeit der laufenden Mitarbeit und der strikten Einhaltung der als Fallfristen definierten (Zwischen-)- Abgabetermine wird nochmals explizit hingewiesen.

Anhang 3

Seminar aus Angewandter Informatik
Sommersemester 2004
R. Mittermeir

Extended Abstracts

Liebe Kolleginnen!
Liebe Kollegen!

Jeder von uns hat schon genügend "Abstracts" gelesen. Daher haben wir in der Regel keine Mühe, ein gutes "Abstract" zu schreiben. Wir müssen lediglich ein paar Aspekte berücksichtigen. Allerdings haben wir, solange wir noch nicht am wissenschaftlichen Qualitätssicherungsprozess aktiv beteiligt waren, kaum Gelegenheit „Extended Abstracts“ zu lesen. Deshalb möchte ich hier kurz auf die Unterschiede zwischen beiden Textsorten eingehen und Ihnen so den nächsten Schritt im Seminar etwas erleichtern.

Funktion des „Abstracts“
Es soll potentielle Leser zu motivieren, doch die ganze Arbeit zu lesen.
Daher der Inhalt:

- *warum* ist, was ich schreibe *wichtig*
- *was erfährst Du* genauer, wenn Du alles liest.

Wichtig (und manchmal schwierig): Ein Abstract ist etwas anderes als ein "Summary" (Zusammenfassung) oder eine "Conclusion" (Schlussfolgerung). Der Inhalt dieser beiden Komponenten eines Aufsatzes geht wohl aus dem Begriff eindeutig hervor und unterscheidet sich von obigem. Aus Lektüre von Abstract und Schlussfolgerung (oder Abstract und Zusammenfassung) sollten Leser ein gutes Bild von dem gewinnen, was sie vom Rest der Arbeit erwarten dürfen. Leider haben wir im Deutschen für Abstract und Summary nur die gemeinsame Übersetzung "Zusammenfassung" was verwirrend ist.

Extended Abstracts
Extended Abstracts sind Zwischenprodukte im wissenschaftlichen Produktionsprozess und dienen der Vorselektion von Aufsätzen, Workshops, Tutorials, Panel Discussions, etc. Daher sehen wir sie als "Konsumenten" des Endprodukts eher nicht mehr und haben etwas weniger Erfahrung damit. Jedenfalls sollte das Extended Abstract auch die Inhalte des eigentlichen Abstracts, also:

- warum ist, was ich schreibe wichtig
- was erfährst Du genauer, wenn Du alles liest (oder hörst)

enthalten.

Doch der Leserkreis ist ein anderer. Es sind nicht die endgültigen Leser/Teilnehmer, sondern die Juroren, die insgesamt eine gute Veranstaltung zustande bringen wollen. Somit sollte der durch

"Extended" gewonnene Raum dazu genutzt werden, Gutachter davon zu überzeugen, dass das, was sie noch gar nicht lesen können

- lesenswert wäre
- der gesamte Beitrag ein positiver Beitrag zur Gesamtveranstaltung sein wird.

Letzteres ist wohl der entscheidende Punkt. Wie er zu bewältigen ist, hängt ein wenig von der jeweiligen Veranstaltung ab. Möglich wären Kriterien wie:

- Kann ein guter Aufsatz erwartet werden (Langfrist-Perspektive)?
  * Ist es interessant / innovativ?
  * Haben die Autoren das Potential und das Material ihre Versprechen einzulösen?
- Kann ein guter Vortrag erwartet werden (Kurzfrist-Perspektive)?
  * Lässt die Qualität der Präsentation des Extended Abstracts auf die für den Vortrag nötige Fokussierung und klare Strukturierung rückschließen?

Insbesondere für letzteres gilt:

- wie passt das zu Erwartende zum Gesamtprogramm,
- wie passt es zur erwartenden restlichen Teilnehmerschaft,
- ist es fokussiert,
- ist es fundiert bzw. wie ist es fundiert,
- klingt es spannend,
- ist es neu, innovativ, kreativ,

  (auch gute Überblicksarbeiten sind nach diesen Kriterien beurteilbar).
  daher auch: was ist das Neue daran (Inhalt, Darstellung, ...)

Somit will ein Gutachter eigentlich aus dem Extended Abstract auf möglichst knappem Raum ausreichend klar erkennen, wie das ganze Werk denn ausschauen wird.

Dieses Ziel der Gutachter wird sicherlich NICHT erreichbar sein, wenn

- Autoren selbst noch nicht ausreichend klar ist, wo er denn "landen" wird,
- sich das Extended Abstract auf ein Motivationskapitel, Zusammenfassungskapitel, ... (irgendeinen Ausschnitt) beschränkt oder durch andere Formen der Unausgewogenheit einen unbeabsichtigt falschen Eindruck vermittelt.

Ich hoffe, dass Sie in Ihrer Vorbereitung diesen Zielen recht nahe kommen und in der Begutachtung hoffentlich auch Extended Abstracts sehen werden, die diesen Vorstellungen einigermaßen gut entsprechen.

Zum Schluss ein Wort des Trostes: Ich denke, ein gutes Extended Abstract zu schreiben, gehört wohl zu den schwierigsten Aufgaben im wissenschaftlichen Publikationsprozess. Aber Sie haben ja auch entsprechend Zeit dafür!

Herzliche Grüße
Roland Mittermeir

Anhang 4
Seminar aus Angewandter Informatik
Sommersemester 2004
R. Mittermeir

Programm der 1. Vortragsrunde
25. Mai 2004, SR 2.42

7:30 Thomas FRANK:
Testen komponenten-basierter Software
7:50 Stefan PERAUER, Robert SORSCHAG:
Effektives Testen objektorientierter Software mit Objekt-Relations-Diagrammen

8:20 Daniela INNERWINKLER, Gunar MÄTZLER:
Mutationentest – Objektorientierte Mutanten für Java-Programme
8:40 Daniel PEINTNER:
Der optimale Software Release Zeitpunkt

9:10 Stefan LAMPICHLER, Petra TESSARS:
Remote Usability Testing
9:30 Marion KURY:
Requirements Engineering in globalen Projekten

Programm der 2. Vortragsrunde
8. Juni 2004, SR 2.42

7:45 Birgit ANTONITSCH, Hubert GRESSL.
Entwicklung von ISO 9000

8:10 Gudrun EGGER.
CMM – Eine Einführung

8:40 Katharina FRITZ, Marina GLATZ.
Zeitmanagement im individuellen Software-Entwicklungsprozess
unter spezieller Berücksichtigung schulischer Aspekte

9:10 Johannes WERNIG-PICHLER, Mario LASSNIG.
Objektorientierte Software-Metriken
9:35 Michael JAKAB, Michael OFNER:
Die „Goal Question Metric“ Methode

Ich ersuche, die Vortragszeit von max. 15 min strikt einzuhalten und die Folien bis Montag 24. Mai 2004 bzw. 8. Juni 2004, jeweils Mittag, in CLAROLINE abzulegen, damit uns auch möglichst viel Zeit für Diskussion erhalten bleibt.

Seminar aus Angewandter Informatik

Sommersemester 2004

R. Mittermeir

Programm der 1. Bakkalaureatsvorträge
15. Juni 2004, SR 2.42

8:05 Kerstin JÖRGL
Management of Requirements Traceability Problems

8:45 Daniela ESBERGER
Aufwandsschätzung am Beispiel COCOMO II

9:25 Ingrid UNTERGUGGENBERGER
Akzeptanztests

Programm der 2. Bakkalaureatsvorträge
22. Juni 2004, SR 2.42

8:05 Mario GRASCHL
Qualitätssicherung bei agiler Software-Entwicklung

8:45 Edmund URBANI
Metriken zur statischen Analyse objektorientierten Source-Codes

9:25 Werner SÜHS
Spannungen zwischen neuer Funktionalität und Benutzergewohnheiten

Programm der 3. Bakkalaureatsvorträge
29. Juni 2004, SR 2.42

8:05 Brigitte GAUSTER
Einsatz und Qualitätssicherung von life-critical Software in der Automobilindustrie

8:45 Ursula DITTRICH
Die formale Inspektion – Eine spezielle Review-Technik

9:25 Roland MITTERMEIR
Abschlussbesprechung

Ich ersuche, die Vortragszeit von 25 bis 30 min einzuhalten. Die Folien sollten bis Montag vor dem Vortrag, jeweils Mittag, in CLAROLINE abgelegt worden sein.